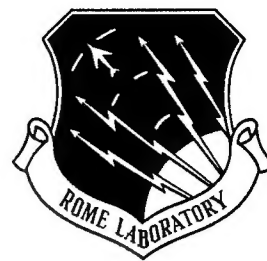RL-TR-97-144
Final Technical Report
October 1997

DIAMONDS: ENGINEERING DISTRIBUTED OBJECT SYSTEMS

Syracuse University

**Evan Cheng, Gary Craig, Nataraj Nagaratnam,
Arvind Srinivansan, Jo Tsai and Paul Tyma**

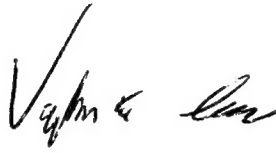*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*

DTIC QUALITY INSPECTED 4

19980209 068

**Rome Laboratory
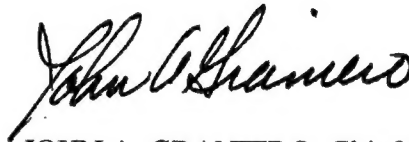Air Force Materiel Command
Rome, New York**

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-97-144 has been reviewed and is approved for publication.

APPROVED:

VAUGHN T. COMBS
Project Engineer

FOR THE DIRECTOR:

JOHN A. GRANIERO, Chief Scientist
Command, Control & Communications Directorate

REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Oct 97	Final Apr 95 - Jun 95

4. TITLE AND SUBTITLE

DIAMONDS: ENGINEERING DISTRIBUTED OBJECT SYSTEMS

6. AUTHOR(S)

Evan Cheng, Gary Craig, Nataraj Nagaratnam, Arvind Srinivasan, Jo Tsai, Paul Tyma

5. FUNDING NUMBERS

C - F30602-95-C-0130
PE - 63728F
PR - 2530
TA - 01
WU- P3

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Syracuse University
Office of Sponsored Programs
113 Bowne Hall
Syracuse, NY 13244-1200

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Laboratory/C3AB
525 Brooks Rd
Rome, NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-97-144

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Vaughn T. Combs, C3AB, 315-330-7155

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report describes DIAMONDS, a research project at Syracuse University, that is dedicated to producing both a methodology and corresponding tools to assist in the development of heterogeneous distributed software. The design is based on cooperative fine-grained objects and a concrete design notation, odl. The mapping of the design and programming model to the run-time computational model incrementally composes tightly-coupled objects into coarse-grained abstractions. The resulting software development process supports continuity from the abstract analysis and design to the concrete implementation and postpones concerns on encapsulation and structure as dictated by the application and/or the problem domain. The report also describes additional support added to DIAMONDS in order to support reliability and resource management.

14. SUBJECT TERMS

Object-Oriented, Distributed Processing, Resource Management

15. NUMBER OF PAGES

72

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

List of Figures

Executive Summary

This effort built off of the infrastructure and results obtained in Rome Laboratory contract #F-30602-93-C0108. As part of the previous effort, a methodology for designing concurrent software based on a fine-grained active object model was developed. This effort extended both functionality and performance of the prototypes developed in the original contract. In addition support for reliability and resource management were explored and prototyped. A new release of the DIAMONDS prototype distributed system is being delivered as part of this contract.

Chapter 1

Introduction

The typical C^3I application has the requirement for high reliability which implies support for software adaptation. Adaptation is required to meet a level of service in the presence of changing computational infrastructure. The changing base may be due to system demands or asset failures (computation, communication, and/or data). Adaptation is also a mechanism available to the infrastructure to better apply and manage its resources to the applied load. Our approach to support and develop adaptive distributed software systems is based on distributed active object.

DIAMONDS, a research project at Syracuse University supported by Rome Laboratory, is dedicated to producing both a methodology and corresponding tools to assist in the development of heterogeneous distributed software [BCSL93]. The design model is based on cooperative *fine-grained*[1] active objects and a concrete design notation, odl[DLea 94, dCLF93]. The mapping of the design and programming model to the run-time computation model incrementally composes tightly-coupled objects into *coarser-grained*[2] abstractions. The resulting software development process supports continuity from abstract analysis and design to concrete implementation and postpones concerns about granularity issues as they effect performance. Instead, developers concentrate on encapsulation and structure as dictated by the application and/or problem domain.

Automation of resource management and performance design issues further enhances software process and products, in particular design portability and reuse. Reliance on an active

---

[1]Our use of the phrase "fine-grained" refers to the requirement that every object obey the same properties, regardless of its size or complexity. Thus, the model applies independently of object granularity.

[2]Coarser applies both to size and mean number of computations per message dispatch.

object model removes the need for explicitly designing special (often somewhat arbitrary) abstractions responsible for the control of the objects. Such *agent* type designs usually result in the suboptimal early commitment to a particular target architecture. Automated composition of fine-grained objects into coarser entities can be performed with reference to a particular target architecture's characteristics, thus normally outperforming manual strategies. More importantly, this can be achieved late in the design cycle.

The DIAMONDS run-time system provides direct support for object clusters via lightweight address spaces. It is ultimately responsible for the construction, placement, mapping, and maintenance of object clusters [BCSL93, BCLS93]. Clusters are dynamically composed into *ensembles* which are placed on physical nodes in the system. Ensembles are analogous to more traditional processes or tasks. The object composition model (design activity) is extended into the run-time environment, where in conjunction with resource management services, collections of object clusters are composed into specific Ensembles (based on directives inserted by the clustering tool).

This run-time composition of clusters into ensembles permits greater flexibility in mapping an application to a variety of architectural granularities. In the unobtainable ideal architecture (unbounded numbers of processors with zero communication latency), each processor node would contain one ensemble, each ensemble would contain one cluster, and each cluster would contain one object. However, filling more than one component in each container provides substantially better performance on contemporary architectures. For example, in the case where multiple clusters are mapped to a single ensemble, communication between objects in different clusters within an ensemble can exploit the fact that they shared the same protection domain. The run-time system would like to have clusters which are large enough to be efficiently managed while at the same time are small enough to be able to exploit the concurrency available in the application and the distributed system. Resource management interacts with the Diamonds run-time to trigger cluster migration to handle major changes in the applications run-time profile and/or the systems resource profile.

This effort concentrated on a number of technical extensions to the preliminary work on DIAMONDS. They include:

- One of the most promising attributes of Diamonds is its ability to provide application resource utilization information to a distributed system resource management framework. A number of ideas and designs had previously been investigated[Cra94], however, prototype(s) (configuration managers, etc) and empirical studies were performed

as part of this study.

Fuzzy logic is employed to coordinate partial, distributed system resource metrics and distributed application metrics toward continual application adaptation [NC96].

- As part of the infrastructure necessary to monitor and report application and system state changes, a general purpose monitoring facility was developed. This monitoring facility provide a multiplex messaging layer over top of sockets.

- Another active part of the DIAMONDS research at this time is the development, design and testing of a set of deferred optimizations to an application's run-time image to take advantage of known locality of method dispatches and interactions. This work involved optimizations to both the static compiler for odl (better SIRF code generation) and the development of a back-end dynamic compiler to map SIRF-code to native binary.

- The other notable activity completed as part of this contract was the review of the concurrency control model of active objects. In particular work on efficient support for *joint actions* within a "guarded action" framework. This effort also worked on providing a new set of algorithms to cope with the distributed deadlock problem.

1.1 Related Work

During the execution of this contract, a closely related project splashed onto the international scene, Java(tm) by Sun Microsystems. Both Diamonds and Java have as a core, a concurrent object-oriented programming language (odl and Java respectively). Java is a passive object model with first class threads, while odl is an active object model. Both Diamonds and Java compile their source code to an intermediate form targeted to a machine independent, virtual machine (SIRF VM and Java VM respectively). Both systems default mode of operation is to have the respective virtual machines interprete the corresponding *bytecodes*. The SIRF VM is a register-based architecture. The JVM is a stack based architecture. Both environments look to "client-side" (run-time) compilation to enhance performance.

Much effort was made by the DIAMONDS team to evaluate where our reseach ideas can fit within the Java framework. There are enough similarities between these two environments to warrant consideration of technical transfer of DIAMONDS research results into the newly emmerging "distributed Java" framework, involving remote method invocations (RMI). Among the earliest work performed by this group, related to Java, was adding the

earliest support for remote object creation and method invocation [NSL96]. Other investigations include:

1. looking at adding Diamonds resource management technology to Java

2. mapping Java to the SIRF VM

3. mapping odl to the Java VM

1.2 Report Organization

The remainder of this final project report describes the individual efforts accomplished. Chapter 2 looks at work on Ensemble and cluster placement algorithms which exploit Fuzzy logic. Chapter 3 describes the Doodle framework developed to supply a common infrastructure for messaging and monitoring within DIAMONDS. Chapter 4 looks at enhancements made to the DIAMONDS compiler including resolution of the DIAMONDS inheritance model. Chapter 5 looks at the early results on the DIAMONDS dynamic compilation framework. Finally, Chapter 6 describes details on joint actions within DIAMONDS and presents a new model and algorithm for detection of distributed deadlocks.

Chapter 2

Compiler Extensions

There were two major tasks related to the DIAMONDS compiler.

1. designing the *inheritance* model and implementing the *dynamic dispatch* mechanism

2. designing and implementing the *register allocator*

Each of the two tasks will be discussed in details in the following sections.

2.1   Inheritance and Dynamic Dispatch

The original *ODL* language report ([DLea 94]) allows both multiple inheritance and single inheritance. While multiple inheritance offers the greatest possible power and flexibility in theory, in practice it can negatively affect on the efficiency of the language. My task was to design and implement an inheritance model that manages to maintain much of the power of multiple inheritance but is also efficient in practice.

2.1.1   Mixin Inheritance

The re-designed inheritance model for *ODL* allows single inheritance and a restricted form of multiple inheritance called *mixin* inheritance. Using mixin inheritance, classes representing completely independent properties are mixed together via inheritance in the target class. It is often the case that the superclasses used in mixin inheritance are often totally useless,

6
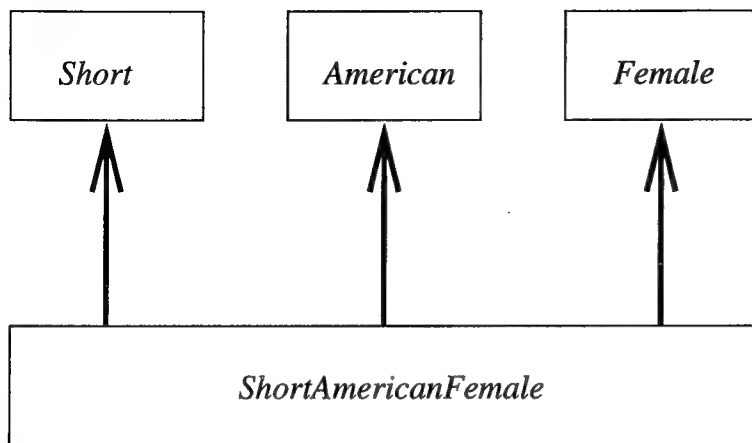
Figure 1: An example of *mixin* inheritance

they may not represent concrete entities. But they can be combined with other classes to form meaningful classes. An example of mixin be seen in figure 1.

In *ODL*, mixin inheritance is used the same way as multiple inheritance. However, only the attributes of the superclasses are inherited by the subclass. The methods of the superclasses are not inherited, but their prototypes (*signatures*) are. The subclass must define the methods declared in its superclasses, and the methods must to the *signatures* of those of the class's superclass. In this respect, mixin inheritance is very much like the interface feature of languages such as *Java*. Furthermore, the subclass may still freely extend itself by declaring additional attributes or methods not found in the definitions of its superclasses. The code segment in program 1 is an example of mixin inheritance in *ODL*.

2.2 Dynamic Dispatch

The design of the method dispatch mechanism in *ODL* and *DIAMONDS* was not changed. However, method dispatch was not fully implemented in *DIAMONDS* to support inheritance, i.e. the virtual machine was not able to resolve the runtime class of a method. Nevertheless, once the *ODL* compiler was modified to support single and mixin inheritance, only trivial changes were made to the method dispatch code in the virtual machine to support inheritance. As it turned out, our system is designed in a way so that most of the tasks for selecting appropriate operation based on runtime class of a method are done during compile time so that the runtime overhead is extremely small. The next section will describe the compiler support for inheritance.

```
        class A                          class B
          local a :  int;                  local b :  int;
          op foo1(x:  int){                op foo3 {
            Print "I am foo1 in A";          Print "I am foo2 in
    B";

            Print "\n";                      Print "\n";
          }                                }
          op foo2() { }
        end;                             end;


        class C mixin A, B
          op foo1(x:  int){
            Print "I am foo1 in C";
            Print "\n";
          }
          op foo2 { }
          op foo3 { }
          op foonew {
            Print "I inherited a from A. ";
            Print "Its value is ";
            Print a;
            Print "\n";
          }
        end;
```

Program 1: An example of *mixin* inheritance in *ODL*

2.2.1 Dispatch Table

The *ODL* compiler generates a separate dispatch table for each class defined. In the case of single inheritance, the dispatch table of the subclass will start out being an exact duplicate of the dispatch table of its superclass. The dispatch table can later be extended and modified as the subclass extends its superclass and override its methods. In the case of mixin inheritance, we use a *coloring-based* allocation scheme to allocate slots in the dispatch table for class methods.

The coloring-based allocation scheme allocates method name offsets in the dispatch table
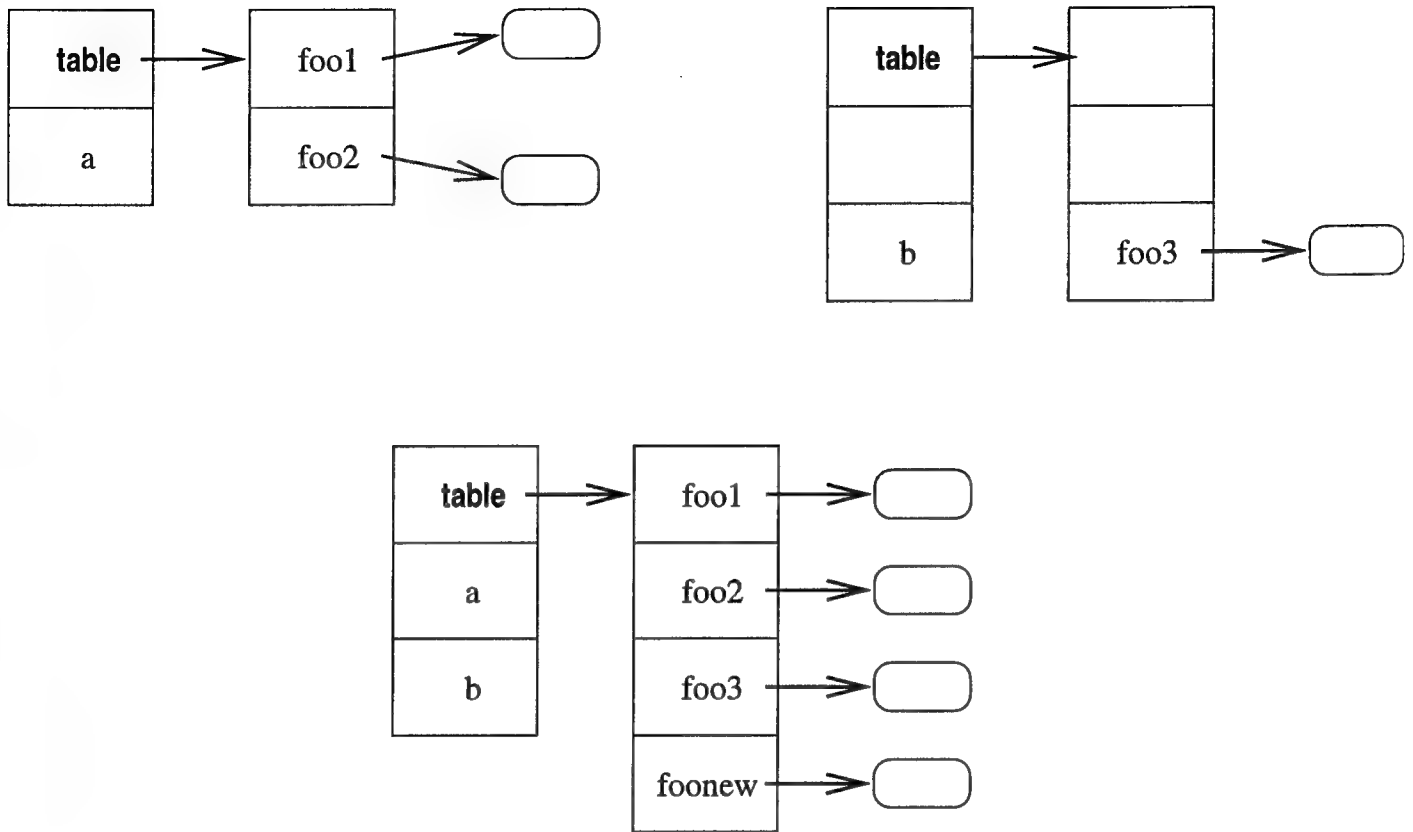
Figure 2: An example coloring-based allocation

globally. To accomplish this, the compiler must understand the *sibling* relationship between two or more classes. Sibliing classes are classes that share a common subclass. In program 1, classes A and B would be considered sibling classes. Given the sibling and inheritance relationships, the coloring-based allocation scheme allocates method table offset globally using one simple rule: *no two methods in the set of sibling classes can share a common table offset.* Figure 2 should help explaining this idea

As it can be seen from figure 2, this dispatch table allocation scheme together with the use of mixin inheritance minimize the time taken to resolve a method dispatch. Consider an object O of declarative class B, regardless of whether the real type of O is B or C, its table offset for method foo1 will be 3. This saves considerable amount of computation during runtime.

2.3   Register Allocator

The original register allocator of the *ODL* compiler assumes an infinite number virtual registers are available. This does not appear to be a problem when the target program of the compiler is interpreted using the *DIAMOND* virtual machine. However, the large number of registers allocated can cause serious problems for the *dynamic compiler* in development.

I re-designed the register allocator using a simple *stacked-based* allocation scheme. It takes advantage of the structure of the three-address format of the intermediate language. For example, the following code segment is typical of the code generated by *ODL* compiler:

```
load   x,   t1
load   y,   t2
t3  := t1 + t2
store t3,    z
```

The register allocator recognizes that the lifetimes of registers t1, t2 do not extend beyond the third statement in the segment. In fact, a register *dies* (is *freed*) as soon as it is used as a operand. By keeping a list of freed registers, the register allocator is able to reuse the dead registers. As soon as a register is used as an operand, it is appended to the list of freed registers. Whenever a new register is to be allocated, the register allocator tries to grab a register off the list. Only when there is not a freed register available, do a brand new register and increment the register count.

The simple scheme turns out to work well for us. It drastically reduced the number virtual register used, often by a factor of 7 to 8 in moderately complex methods.

Chapter 3

DIAMONDS Runtime

Diamonds environment consists of the static analysis tool, the run time, central resource manager and the information repository of the system and application metrics. The components of the Diamonds environment execute the application code with available system and application information. The runtime groups the objects to individual clusters and clusters into ensembles. To achieve better application performance and system throughput, the runtime and the resource manager co-operatre in mapping these ensembles to physical nodes. Using the available system metrics, initial ensemble placement and initial cluster distribution is accomplished (more details are provided in [NC96]).

3.1   Diamonds Super Server

Before executing any application, the Diamonds SuperServer is started as a daemon process. It binds itself to a socket and listens for service requests from the Diamonds runtime. For every request it receives, it checks the file "dssd.conf" to check whether or not the service is available. If it is available, the appropriate process is started to provide the necessary service.

For example, the file dssd.conf (Diamonds SuperServer Daemon - Configuration file) contains the following line:

startEnsemble /serval2/users/nataraj/diamondsBuild/bin/sun4_solaris/ensemble

The first word 'startEnsemble' is the service name and the following string is the process to be started when the SuperServer receives the request. The daemon is first started by issuing

11

the following command:

DIAMONDSd -f dssd.conf

3.2   Program Startup

Now that the SuperServer is ready to service requests, the application can be run by the runtime. The Diamonds distributed runtime (drt) is executed by issuing the following command:

drt -H <host1> ..<hostn> -f <sirf-file-name>

for example,

drt -H serval lion -f /diamondsCode/queens.sirf

The runtime starts executing the application by first initiating the application monitoring tool (odlview.tk). The monitoring tool listens to a socket for messages from ensembles. It processes the messages and displays the appropriate information. The next step is to gather system metrics from the specified hosts. The metrics are used to decide on the initial ensemble placement and appropriate cluster distribution.

3.3   Collecting System Metrics

The number of ensembles created on a host is equal to the number of CPUs configured on that host. The EnsembleGenerator, responsible for initial ensemble creation, collects the physical system metrics of each of the specified host (on the command line). The EnsembleGenerator sends a request to the SuperServer to obtain the system metrics of each of the hosts. The "getStats" is a registered service of the SuperServer, which collects the metrics information. The details of a host are provided by the SuperServer running on that particular host. The collected metrics include:

- Number of CPUs configured for that host

- CPU speed

- Physical memory capacity

- Load average on the system at that instance of ensemble creation

12

The EnsembleGenerator creates an ensemble for every processor after obtaining the information about the hosts. The number of ensembles on a host is equal to the number of configured CPUs on that machine. The other host metrics are used to predict the anticipated throughput (AnT) of the host which is used in the initial distribution of clusters among the ensembles.

3.4   Using Fuzzy Rules to predict AnT

Using the metrics collected from the system, the anticipated throughput of each of the host is calculated using the following set of fuzzy rules:

- If cpu_load_per_processor is High, then Ant is Low

- If no_of_cpus is Low AND cpu_speed is Not-High, then Ant is Low

- If cpu_speed is Medium AND no_of_cpus is High, then Ant is Medium

- If avail_memory is Medium, then Ant is Medium

- If cpu_speed is Very-High OR no_of_cpus is Very-High, then Ant is High

Applying these fuzzy rules to the machines the different throughput capacities for those machines can be calculated. The fuzzy min-max method is used as the inference scheme to reduce the truth of a consequent fuzzy region. The correlation minimum method is used to restrict the height. The defuzzification of the resultant fuzzy regions are accomplished using the method of composite moments. By defuzzification of the resultant fuzzy region of ensembles, the least to heavily loaded ensembles can be determined.

3.5   Dynamic Cluster Distribution

The runtime initially creates one cluster to start the program. This cluster has a Program object that is responsible for the exeuction of the application. Requests for creating more clusters are generated when the hint *asSeed* is encountered. For every object created with the hint, a request for a new cluster gets generated. As more requests for creating clusters

come in, they are distributed across the different hosts proportional to their AnT. Within a host, there may be more than one ensemble, if the host is a multi-porcessor machine. In this case, the cluster assigned to that host is allocated to one of those ensembles in a round-robin fashion.

3.6 Application Metrics and Dynamic System Metrics

After the initial distribution, the application is executed. The details of the inter-cluster and intra-cluster communication are maintained at the ensemble level. Whereas the inter-ensemble and intra-ensemble communication metrics are derived from these inter/intra-cluster metrics and are available at the Program level. The dynamics of the application reflect the varying communication pattern among the clusters and hence the ensembles. High communication requirement between ensembles calls for the reconfiguration of the clusters, so that the intra-ensemble communication should be increased and the inter-ensemble communication should be reduced. This will help in enhancing the overall performance of the distributed system under the Diamonds environment.

3.7 The Future

Presently, the static metrics and the system load are used in the decisions concerning the initial placement of ensembles and distribution of clusters. The dynamic metrics, though collected, are not yet used. These dynamic metrics will be used in making decisions concerning the migration of clusters between ensembles, resulting in performance improvement of the Diamonds environment.

Chapter 4

Doodle Messaging Framework

A distributed system, by its nature, introduces truly asynchronous operation invocation. A typical application on a distributed system is decomposed into sub-tasks, and these sub-tasks are executed on different computers. Information about these sub-tasks must be linked to the *source* application. Maintaining per-application context information is an important function of a distributed system. Distributed systems need to provide a mechanism for applications to send informational messages as well as a mechanism for collating these messages and presenting them in a relevant form. Such information needs to be available from any node on the network. Policies/mechanisms to facilitate this information flow need to be consistent across all the nodes on the network. Distributed systems need to provide application developers and system administrators with tools that help manage the complexity of working in a distributed environment.

Factors such as heterogeneity, inherent concurrency and communication delays, make monitoring and debugging programs in a distributed system a challenging task. In a distributed computing environment, processes may update their states independently or in response to the actions of other processes. By nature, distributed systems are non-deterministic and replicating a sequence of actions is one of the main concerns of system designers/administrators.

Service requests in distributed systems are typically serviced by system daemons running on each node. It is hard to determine the process environment of these daemons, such as whether it is attached to a terminal or not. Hence, distributed applications cannot make assumptions about writing messages to output streams such as standard error or standard output. Monitoring the state of an application becomes tedious, if each sub-task of the application writes information to a "private" file. Having all applications write to a single

file brings up synchronization and serialization problems. Centralizing the logging activities of the distributed event monitoring framework serializes access to output devices such as printers, files and avoids the issues and problems caused by different processes writing to the same output device simultaneously.

An effective logging infrastructure in a distributed system will help facilitate the following:

- Building tools on top of this logging mechanism to help system administrators *monitor* the state of the system.

- Helping application developers visualize how their application is executing in the distributed environment.

- Analyzing the pattern of communication between the objects that comprise a distributed application.

- Identifying the cause of failures by analyzing the state of the system at the time of failure and after the failure.

4.1   Distributed Monitoring Framework - Goals

In the design of this framework, goals were set to address each of the problems faced by event monitoring systems on an individual basis.

- **Generic:** The framework should be not be too dependent on the specifics of a particular implementation of a distributed system. This independence will increase the reusability of the framework.

- **Object-oriented design:** The software design process should use object-oriented design methodologies. Object oriented languages offer three basic facilities for organizing a collection of classes: inheritance, aggregation and parameterization. Class inheritance is a very useful structuring principle for classes. C++ encourages abstraction by hiding the private implementation details behind a public interface. Virtual functions in C++ enable the redefinition of the semantics of concrete method invocations from a more general abstraction. The use of C++ templates is also a powerful tool in designing a powerful set of classes.

- **Presentation:** A graphical front-end to the monitoring framework should present accurate, up-to-date information about the status of the system. The information must be presented so that it reflects the context of the application in which the event occurred.

- **Portability:** Due to the heterogeneous nature of a distributed system, the code that implements the logging mechanism must be portable across different operating system platforms. The data representation of messages should be independent of a particular operating system platform.

- **Extensibility:** The framework should be designed such that extensions can be made to it by adding new classes without compromising the integrity of the framework

- **Simplicity:** Using the logging mechanism from within an application should be via a simple programming interface. The graphical user interface should be easy to use and should be designed such that the various parts of the system can be monitored by simply pointing and clicking the mouse pointer on the graphical view of the system. Integrating the framework into an existing implementation of a distributed system should be seamless and not require a redesign of the components distributed system.

- **Complete:** The framework must provide a family of domain-independent classes that can be used to effectively monitor the events in the distributed system. The information collected and presented by the monitoring framework should be complete.

- **Interoperability:** In a heterogeneous environment, a distributed application may run on client desktops, servers, and mainframes. Applications must be able to compensate for differences in data representations between the different hardware architectures that participate in the application. The framework must support the requirements of interoperability as the computers in most distributed computing environments have different types of hardware architectures.

- **Monitor Diamonds:** The framework must be integrated with the Diamonds distributed environment. Using the framework, one should be able to monitor the Diamonds environment.

4.2   Distributed Monitoring Framework - Effort

In this section we discuss the design of the distributed logging framework and some of the major design decisions that were adopted and the factors that influenced these decisions

Context information about events are sent to the monitoring system. Each "packet" of information describes the context in which the event occurred. A simple yet extensible message format was designed. A message is composed of a message header section and a message content section. The message header contains the essential information required of every message (such as message length, message type etc). The message content section contains additional information that is specific to the type of message. This message abstraction was encapsulated in a set of hierarchical classes and the hierarchy can be extended to support custom formats for the message content.
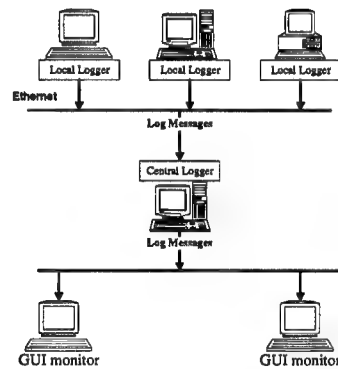

Figure 3: Monitoring framework - system model

The distributed monitoring framework is based on the client-server model for developing

distributed applications. The approach adopted by the monitoring framework was to have logging server daemons running on each computer and a central logging daemon running on one computer on the network. The local logging servers establish socket connections with the central logging daemon. All applications on a machine send log information to the local logging server running on that machine, and the local logging server daemon forwards the messages to the central logger. Figure 3 shows the system overview of the distributed monitoring framework.

The framework was developed by building a forest of classes related by lines of inheritance. The decomposition of the problem domain into classes was done after extensive analysis into the characteristics and functionality of each entity in the distributed system. Each cluster of classes is placed into its own category. Abstract base classes were used to capture the relevant public design decisions about the abstraction. Concrete classes were used to represent a particular entity in the infrastructure of the framework. Building upon a hierarchy of classes assures that the sibling concrete classes share the common behavior. Using design patterns for modelling classes is a well-proven route to developing and packaging reusable software components. Distributed application developers can monitor application-specific events by inserting calls to the logging API, at appropriate sections in the application.

A graphical front-end to this event monitoring framework was developed to provide visual feedback to the user and to enhance the overall usability of the system. The graphical front-end parses the message header to extract context information and display this information in a simple yet attractive user-interface.

The computation of a distributed application in the Diamonds environment is divided into one or more ensembles containing one or more clusters. To monitor the state of a distributed application, the state of the individual clusters and of the ensemble as a whole must be monitored. The "birth" and "death" of clusters and ensembles was monitored by inserting the code to communicate these events to the event monitoring mechanism into the constructors and destructors of these objects. Ensembles and clusters are identified by unique integers that are assigned by **Program**. Objects are identified by using a combination of its cluster identifier as well as its integer identifier. By using these unique identifiers, the placement of ensembles and clusters among the various nodes of the network was presented using a visual interface. Figure 4 shows a screenshot of a component of the graphical interface developed to monitor Diamonds.
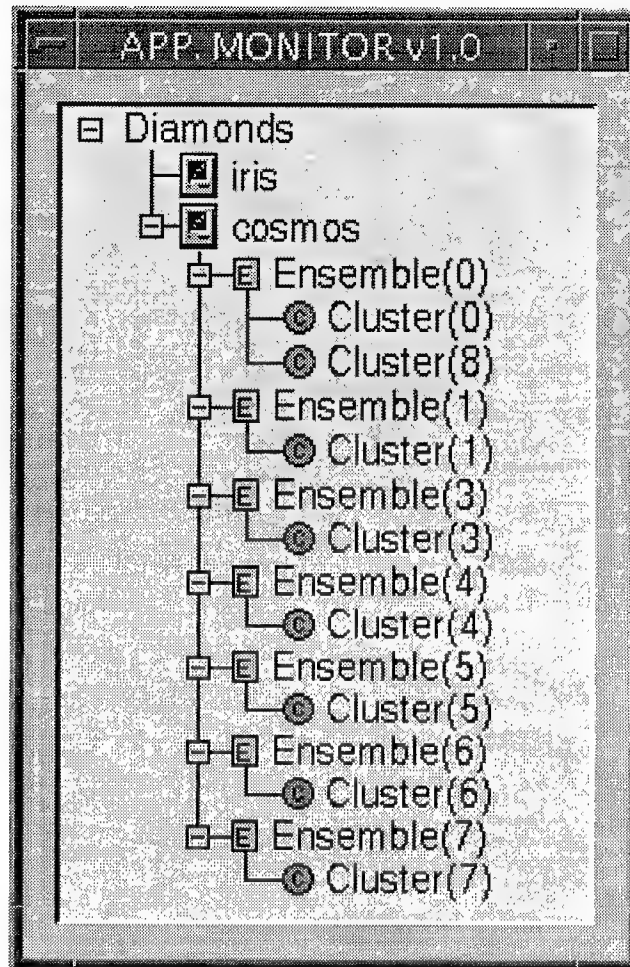
Figure 4: Diamonds application monitor - screenshot

In the Diamonds runtime system, modules that run on each node on the network, continuously collect metrics such as processor load, virtual memory usage. These modules could use the logging framework to route these metrics to the central logger that would then present a comprehensive system-wide view of the distributed system.

4.3 Distributed Monitoring Framework - Conclusion

A prototype of the framework was developed to monitor the Diamonds distributed environment. Object-oriented design methodologies such as class inheritance, aggregation and design patterns were used to develop a hierarchy of classes. These classes represent the various event monitoring modules at different levels of abstraction. Data is written to and

read from the physical channels of communications in network byte order format, to account for potential byte order differences between different hardware architectures. The *select()* demultiplexing mechanism is wrapped within a class that has a simple public interface. The framework uses efficient interprocess communication mechanisms such as FIFOs for intra-host communications and sockets for inter-host communication. The implementation details of event demultiplexing on multiple file descriptors and inter process communication are hidden behind classes with well-defined public interfaces. Encapsulating these complex programming paradigms in well-defined classes reduces the risk of programming error and enhances the portability of the framework as the private implementation details can be changed without affecting any applications that use the classes. The graphical user interface to the monitoring mechanism, uses an application identification integer and the event timestamp to order the events in the context of a particular application.

Incorporating the framework to monitor Diamonds involved extending the framework to capture some of the abstractions specific to Diamonds. We were able to do this with little effort. Invoking calls to the monitoring APIs from within the Diamonds code was done easily. We were able to customize the graphical front-end to visualize the Diamonds abstractions (clusters/ensembles) and to passively monitor Diamonds. By incorporating the framework into Diamonds we were able to test and validate the goals of this research effort.

The simplicity, portability and extensibility of this monitoring mechanism makes it a very useful tool that can be used by application developers and system administrators in any distributed environment.

Chapter 5

Dynamic Compiler

5.1 Target Architecture:

HP PA-RISC

Long term goals of the project were to develop an environment for DIAMONDS execution which began as interpreted but was progressively compiled as runtime logic determined execution 'hot spots'. Initial development goals (the focus of this work) was to develop a straight dynamic compiler for SIRF code into HP PA-RISC machine code. The incremental compilation logic could be added later (the initial design was directed to facilitate this ability).

Given the heterogenous environment DIAMONDS is built upon and the inherent machine-specific nature of this project, extra care was taken to ensure high levels of abstraction of specific machine characteristics. Most importantly, this included virtual machine memory organization, instruction sizes, and SIRF instruction mapping. In essence, the conversion from SIRF to machine code was nearly a straight translation (i.e. one sufficiently large CASE statement could perform the task). Consideration of the overall design represented the more interesting (and challenging) aspects of this work.

5.2 Design Considerations

Some considerations were shaped by knowledge gained after a significant piece of the work was completed. Many resulted from attempts to abstract the idiosynchrasies of the PA-RISC and HP-UX environments. Subsequent architectures are guarenteed to induce further changes.

- Initial implementation was to have a "code" space and a "data" space for the dynamically created environment. The memory space protection mechanism in the HP environment proved to foil this plan. In consideration of this protection and the risk of even further vehemence by future architectures, the design resolved to allow the code and data spaces to share one physical allocation and perform memory protection/organization in the dynamic compiler. For general usage, the model will be simple, but more advanced techniques (such as recursion) could incur considerable memory handling effort and may outweigh the simplicity of this model.

- The initial attempt used the existing interpreted environment as a base and added compilation functionality. This method was desirable because the final product (i.e. including incremental compilation) would exist within this framework. However, in consideration of a initial goal of creating a straight compiled environment, the architecture of the interpreted system became cumbersome. So, selected elements of the work were extracted to create a standalone dynamic compiler.

- As with the interpreted environment, the design would be SIRF-code-centric. This would allow for specific implementations of a SIRF code across architectures (i.e. implementation of the SIRF Add instruction as completely different instructions on different architectures.).

- Initial implementation would work in distinct steps to help debugging. This involved, reading the Sirf code, compiling the Sirf code, executing the program. Final implementation would be less distinct to help reduce compilation lag times (i.e. compilation of data streaming across a network takes minimal time compared to the network delay)

- Given the initial wholly compiled environment, method calls would be straightforward. Taking advantage of the architectures Branch-and-link instructions would inherently use the system stack (or comparable subroutine calling device) to provide the needed call and return functionality. However, once methods can run interpreted or compiled,

method calling will become more difficult. The problem largely stems from compiled and interpreted methods intermixing calls. In addition, the prospect of a call to a method which may have changed status (i.e. gone from interpreted to compiled) would further exacerbate the issue. Consider the following scenarios:

1. Interpreted method A calls interpreted method B - no problem (i.e. current DIAMONDS organization). In other words, interpreter handles state saves and return of control.

2. Interpreted method A calls compiled method B - again no problem, B returns and interpreter regains control, knowing where it left off.

3. Compiled method A calls interpreted method B - can be dealt with. Method A flags the interpreter that it needs a method call (i.e. its not done executing). Interpreter can then use a fat stack (a stack which holds extra return info) to execute method B and pass control back to A.

4. Compiled method A calls compiled method B - this is a problem, since this call could have been encountered before while B was still interpreted. Using the system stack and making this a "normal" and fast system subroutine call is not conveniently feasible. This would require knowing all methods where method B is called from and changing all compiled versions of those methods (or changing the call code for each subsequent call to B as it is encountered). The chosen solution is to allow A to return control to the interpreter and flag the interpreter to call B then return control to A. Obviously, this unfortunate overhead is just that, unfortunate overhead.

This methodology will keep the interpreter in control a good deal of the time. This will provide an easy implementation of method usage counts or other logic needed to determine method usage As in the above example, compiled to compiled method call statistics are obviously not particularly useful in determining heavily used methods which are candidates for compilation (since both methods are already of that status). This would, however, allow for statistics needed to determine candidates for incremental recompilation with optimization in mind (i.e. as in the Self incremental compiler).

- Actual method call code (not derived from Sirf) needs to be generated and is purely machine/OS specific. Because of this, the sequence of machine instructions must be determined for each architecture. A level of abstraction only provides a known place in the code to put the sequence. No greater considerations to this code, beyond it

24

being "some instructions at the front and end of each method" has been discovered. Hopefully, this would remain true on subsequent architectures.

- Register allocation was an important initial consideration. SIRF by nature, never reused registers and madly went about allocating new registers continuously. A real architecture required allocation of existing, unused registers. Fortunately, Evan implemented reuse of registers while generating SIRF code (i.e. this is where it belonged, not here) and eliminated and immediate danger. It is still undebatable, however, that large programs will soon overflow all available registers and the dynamic compiler will require register spillage handling.

5.3   Implementation Benchmarks

Sirf to Machine Expectedly, SIRF's relatively granular design mapped well to actual machine instructions. Often, the relationship was one SIRF to one actual machine instruction. Facilities were provided for any number of machine instructions that were needed to model a single SIRF instruction. Possible future implementations could consider a "Many to one" implementation of several SIRF codes represented by a single (or smaller set of) well orchestrated CISC instruction depending upon the architecture. This optimization would be complex and was considered at this time.

BackPatching All Jump instructions required backpatching after actual memory were determined. If all SIRFs translated in a one-to-one relation with actual machine instructions a backpatching step could be done on-the-fly, but this is never expected to be the case. All instructions requiring backpatching are remembered in a data structure and subsequently patched after the entire method has been compiled. Because of this nature, backpatching will always represent a critical section of code that cannot run simultaneously with compilation or execution. Its lag will be trivial, but always present.

Method Calls Any fancy method calling considerations are simply replaced with code to return to the interpreter. The layout of the code space includes bytes for "communication" between the compiled method and the interpreter. The format is as:

0000 - 0000 Message to Interpreter. Zero indicates method complete, nonzero value indicates method number this method wishes to call. This is effectively the communication or "command" byte between the method calls.

0001 - 0001 Pointer to instruction to return to after call outside is done. Since the method call will need to return here, the actual address is provided. Initially, it would seem that simply the "next" instruction would be the return point, however, differences in call procedures and allowance for subsequent architectures makes the extra task of specifying this location worthwhile.

0002 - 0002 Return Parameter (this may change)

0003 - XXXX Machine Instructions - i.e. the code itself. The subsequent location (XXXX) is variable since the number of instructions of every method is variable.

XXXX - YYYY Method Parameters - Parameters are placed beyond the code starting where the code left off (i.e. offset XXXX) and continuing again for a variable length (until YYYY).

YYYY - ZZZZ Data Space - Finally, data values are stored.

It is the responsibility of the compiled code to fill in appropriate message slots before returning to the interpreter. Obviously, this entire architecture exacerbates the already significant problem of method call overhead. Inlining could be implemented to eliminate the problem.

5.4 Usage

As has been discussed, the system is not incorporated into the current DIAMONDS framework and exists as a standalone binary. The executable is passed a SIRF file which it compiles and runs. For diagnostic purposes, display of the generated hex machine codes is provided. To review program execution, a dump of the CODE/DATA space is given before and after execution.

```
op main {
local i:int;
local j:int;
local k:int;
k := 1;
j := 1;
```

```
while (j<99) do
  i := 8191;
  while (i>3) do
    i := i - 1;
  end;
 j := j + 2;
 k := k + 8;
 end;
 }


34070002  LDI 1(0),%r7
68A70018  STW %r7,24(0,%r5)
34080002  LDI 1(0),%r8
68A80010  STW %r8,16(0,%r5)
  OR 0,0,0
48A90010  LDW %r9,16(0,%r5)
340A00C6  LDI 99(0),%r10
  COMCLR,>= %r9,%r10,%r11
  LDO 1(0),%r11
  COMIBF,=,0 1,%r11,31
340C3FFE  LDI 8191(0),%r12
68AC0008  STW %r12,8(0,%r5)
  OR 0,0,0
48AD0008  LDW %r13,8(0,%r5)
340E0006  LDI 3(0),%r14
  COMCLR,<= %r13,%r14,%r15
  LDO 1(0),%r15
  COMIBF,=,0 1,%r15,21
48B00008  LDW %r16,8(0,%r5)
34110002  LDI 1(0),%r17
  SUB %r16,%r17,%r18
68B20008  STW %r18,8(0,%r5)
  BL,= 0,0
  OR 0,0,0
48B30010  LDW %r19,16(0,%r5)
34140004  LDI 2(0),%r20
  ADD %r19,%r20,%r21
68B50010  STW %r21,16(0,%r5)
48B60018  LDW %r22,24(0,%r5)
34170010  LDI 8(0),%r23
  ADD %r22,%r23,%r24
68B80018  STW %r24,24(0,%r5)
```

```
BL,= 0,0
OR 0,0,0
```

```
Dump of CODESPACE:
0000: 00000000 00000000 00000000 073A1384 08040241 081E0244 6FC10080 34A50180
0040: 08A00246 34A50028 34070002 68A70018 34080002 68A80010 08000240 48A90010
0080: 340A00C6 0949588B 340B0002 8D6220D0 08000240 340C3FFE 68AC0008 08000240
00C0: 48AD0008 340E0006 09CD688F 340F0002 8DE22030 08000240 48B00008 34110002
0100: 0A300412 68B20008 E81F1F9D 08000240 08000240 48B30010 34140004 0A930615
0140: 68B50010 48B60018 34170010 0AF60618 68B80018 E81F1EFD 08000240 08000240
 Passing Control to CodeSpace...
 Interpreter regained control...
Dump of CODESPACE:
0000: 00000000 00000000 00000000 073A1384 08040241 081E0244 6FC10080 34A50180
0040: 08A00246 34A50028 34070002 68A70018 34080002 68A80010 08000240 48A90010
0080: 340A00C6 0949588B 340B0002 8D6220D0 08000240 340C3FFE 68AC0008 08000240
00C0: 48AD0008 340E0006 09CD688F 340F0002 8DE22030 08000240 48B00008 34110002
0100: 0A300412 68B20008 E81F1F9D 08000240 08000240 48B30010 34140004 0A930615
0140: 68B50010 48B60018 34170010 0AF60618 68B80018 E81F1EFD 08000240 08000240
Dump of DATASPACE:
0000: 08000240 00000000 00000000 00000000 00000000 00000000 00000003 00000063
0040: 00000189 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0080: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00C0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0100: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0140: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0180: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
01C0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

```
execution completed
```

As defined by the memory layout, the data space exists behind the code space (in the same piece of allocated memory).

5.5   Observations

Performance boosts have been expectedly dramatic. 1000x speedup on tight loops (obviously an estimate) was observed. As the framework becomes more complex, actual run times of just code will slow. This will become apparent with highly abstracted and modular code. In

any case, dynamically compiled code should always outperform the interpreted environment by a wide margin.

5.5.1 Limitations

- The PA-RISC instructions are interestingly designed with a limitation of 0x1FFF for constants within the instruction. For initial simplicity, this was acceptable and used as is. Of course, further implementation will require the ability to surpass such a limit. Ways around this are straightforward but tedious and will be implemented as testing requires.

- Register spillage logic still needs to be added.

- Implement 'selective' compilation and the existence of a run that is part interpretive and part compiled

- HP specific optimizations, such as fill of delay slots and possible reordering of instructions to prevent pipeline stalls.

- Since the code will be half compiled/interpreted we can no longer rely on the interpreter's call stack, nor on the systems stack. Currently we rely on a fat stack providing additional information for call returns. This stack will require further enhancement as its integrated into the interpreted environment.

5.6 Conclusions

As was expected, the largest hurdles of this work were unexpected. Translation from SIRF to PA-RISC was straightforward given SIRF's comparative high-level with respect to actual machine code. Details involving memory management, and the prospect of the C++ language populating an array with values, then passing control to it (something it was not seemingly designed to do) required tricks to prevent the OS from calling foul.

Unquestionably, further work would solidify this work. The compiler remains fragile. In retrospect, PA-RISC was a good choice for an initial architecture because it contains a reasonable number of registers and its instruction set is representative of the other architectures.

Chapter 6

DIAMONDS Guards

The ODL language [DLea 94] is developed as part of the Diamonds project [BCSL93] at Syracuse University. The notion of ODL guard method describes both the local guard and joint-action guard activities. In this work, the Exclusion notation is introduced to uniquely describe a flexible lifetime of exclusive access, with regard to a guard activity, to an instance of an ODL class and it does enrich the expressive power of ODL guard method. The enrichness on ODL guard method is demonstrated in the subsection of Effectiveness of Exclusion Facility. Meanwhile, the Exclusion notation yields a flexible lock granularity of exclusive access to either an object or a method with the state(s) of its guarding object. The two above-mentioned flexibilities are absent in both DisCo specification language [Tampere 90] and concurrent object-oriented languages such as JAVA [SunMicro 95], μC++ [PBuhr 95] and ADA [ABurns 87].

Along with the use of the Exclusion notation to ensure the consistent post-condition of guard evaluation and the consistent semantics of guarded transition(s), the liveness issue in a distributed object system should be considered as well. "Deadlock handling is difficult in distributed system because no site has accurate knowledge of the system state." [MSinghal 89] Various published notions [KChandy 83, MRosler 89, BSanders 89, MSinghal 89] are proposed to observe deadlocks in terms of process in a distributed environment. Yet, the observations are later proved either incomplete or error-prone. In this work, a novel notion of deadlock reasoning in distributed object systems is proposed by devising some unique notations and assertions to capture a minimum set of persistent temporal information of casual relations among a dynamic group of objects. Since a deadlock can be concluded upon a minimum set of persistent temporal information, this notion of deadlock descriptions yields a more

pragmatic and intelligent deadlock detection algorithm over that of [JBrzezinski 95] ,and it excludes the false deadlocks described in [AKshemkalyani 94, MRosler 89, BSanders 89] as well. Furthermore, we suggest an effective approach with a decentralized management to implement the corresponding detection algorithm, and this algorithm retains scalability as the underlying system evolves to grow, in terms of the number of existing objects. Besides, a distinct deadlock scenario caused by object aliasing [JHogg 92] is introduced with respect to joint-action activity [dCLF93, Tampere 90]. Some inexpensive corresponding deadlock avoidance mechanisms are addressed regarding joint-action activity.

"It has been pointed out that inheritance and synchronization constraints in concurrent object systems often conflict with each other, resulting in inheritance anomaly where re-definitions of inherited methods are necessary in order to maintain the integrity of concurrent objects". [SMatsuoka 93] In order to target the reuse of a guarded method code, a selective aggregation mechanism with inlining technique for a coding scheme of nesting a guarded method code is proposed as an efficient solution for inheritance anomaly.

A very much optimized event-driven reflective model is devised for implementing the ODL guard method. So as to minimize the runtime overhead of an event-driven activity, we have devised a unique incremental inter-method dependency mechanism that statically provides each potential dynamic behavior of a method with better information about the inter-method dependency set. Meanwhile, the static analysis *rule 5* is devised to support a message-grouping mechanism in order to reduce the frequency of the context switching, the communication latency, and the message process latency for a joint-action guard evaluation. In addition to the above-mentioned optimization techniques, some runtime components are designed by using C++ [BStroutStrup 93] to implement the synchronization scheme for joint-action activity. This design approach fully eliminates the overhead of generating every proxy object for the receiving and disposing of a returned reply of each two-way, inter-object communication. In this optimized event-driven reflective model, a fully decentralized man-agement is cast to coordinate the joint-action activities among distributed objects; therefore, each participant object engaging in a joint-action activity remains autonomous. We believe that this model will also be very suitable and effective for high-performance μ-processor architecture such as the message-driven processor.

Furthermore, we demonstrate the effectiveness of the optimized reflective model by showing the measurements of optimized concurrent evaluation vs. serial evaluation on a joint-action guard. In the subsection of conclusion, we further indicate that some of the devised mech-anisms are applicable to the development of open systems environment. An open system

environment is characterized by its extensibility to support the dynamic composition of dispersed components on-the-fly, as well as its reactivity of event-driven transactions.

6.1   Syntax of the ODL Guard Method

The partial ODL EBNF for the ODL guard method is presented in Fig. 5. The **Exclusion** notation uniquely expresses a flexible lifetime of mutual exclusive access to an instance of an ODL class with regard to a guard activity. Two reserved words, "transitExclusion" and "evaluateExclusion", are added to the ODL language.

Accept:Exclusion when **Exp** then **[Op]*** **ElseAccepts** end
ElseAccepts: [elsewhen **Exp** then **[Op]***]* else **[Op]*** end

When: Exclusion when **Exp** then **OpSpec ElseWhens** end
ElseWhens: [elsewhen **Exp** then **OpSpec**]* else **OpSpec**

Exclusion: [transitExclusion(**GIDs**);]^i [evaluateExclusion(**GIDs**)]^j |
[evaluateExclusion(**GIDs**);]^i [transitExclusion(**GIDs**)]^j
where **GIDs** is either a non-empty or an empty list of object IDs,
$i, j \in \{0, 1\}$ and i, j denote the frequency of recurrence

Figure 5: The EBNF of the ODL Guard Method

In Fig. 5, each bold word beginning with a capital letter represents a specific notation of the ODL language; in addition, every non-italic-bold word is a reserved word of ODL language. Each **Exp** notation describes either an atomic guard or a composite guard. A composite guard can be decomposed into several atomic guards. Each atomic guard describes the expected state of an object. A composite guard can be the composition of local state(s) and the state(s) of other object(s). When the state(s) of other object(s) is described in the **Exp** of **When** notation, it is termed the "joint-action" guard; otherwise, it is a local guard.

6.2   Semantics of the ODL Guard Method

The ODL guard method permits nest guard. The **Accept** notation denotes that an incoming method call is served only upon the evaluation that its associated guard is true. The **When**

notation can be used for describing both conditional critical region (CCR) and joint-action activity among distributed objects. If an ODL guard beginning with a "when" clause and ending with a "pend" clause is evaluated as false, then the corresponding transition(s) will be delayed until the guard is found to be true. Otherwise, if an ODL guard begins with a "when" clause and ends without a "pend" clause, no reevaluation will occur and a non-blocking effect is presented. The condition in each "elsewhen" clause is interpreted to include the negation of all preceding conditions. Upon the evaluation that an outer-level guard is true, the following inner-level guard is evaluated. If the inner-level guard is evaluated to be false, the corresponding guarded transition(s) will not be pursued. The guarded transition(s) is delayed until a reevaluation on the inner-level guard is found to be true. The "evaluateExclusion()" and "transitExclusion()" clauses annotate the atomicity control over the associated list of parameter object(s) for a corresponding guard evaluation and its associated consequent firing of guarded transition(s), respectively. The Exclusion facility of an outer-level ODL guard annotates only its corresponding level of guard(s) and guarded transition(s). While guarded transition(s) is delayed, the atomicity control on the associated list of objects is not in effect until a reevaluation of the corresponding guard.

6.3 Effectiveness of the Exclusion Facility

```
op A1{
  while true do
    a.wLock; b.wLock; self.rLock;
    if(a.M_a(...)+b.M_b1(...)+b.M_b2(...)+Local(...)) > 0
    then
      self.rRelease; ... do something;
    else self.rRelease end;
    a.wRelease; b.wRelease;
  end }   Example 1
```

```
op A2
  transitExclusion(a,b); evaluateExclusion(self);
    when(a.M_a(...)+b.M_b(...)+b.M_b2(...)+Local(...)) > 0
    then
      { ... do something; }
    else pend
  end
```
Example 2

The Exclusion facility is introduced to off-load the ODL programmer's coding work from synchronization hacking among distributed objects. The expressive power of the Exclusion facility is demonstrated with a comparison between the ODL code A1 of Example 1 and the ODL code A2 of Example 2; both codes of A1 and A2 perform the exactly equivalent joint-action activity. The $a.wLock$ is a statement to lock the object a for exclusive access of write. The $self.rLock$ is a statement to lock a local object itself for exclusive "read" access. The $b.wRelease$ is a statement to inform the object b of being released from the exclusive

"write" access privileged to the task of Example 1; $self.rRelease$ is a statement to inform a local object of being released from the exclusive "read" access privileged to the task of Example 1. Moreover, the Syracuse Intermediate Representation Form (SIRF) of code A2 of Example 2 is very much optimized. At compilation time an ODL code is translated into a form of intermediate code (i.e., SIRF).

6.4   Liveness

The liveness defines that each expected event for firing every delayed activity in a distributed system will eventually happen. If some of the expected events never occur, then it gives rise to deadlock.

6.4.1   Deadlock Detection

In this subsection, a novel notion to deadlock reasoning in a distributed object system is devised. Some corresponding notations and assertions are defined to precisely capture the temporal logic of casual relations among a dynamic group of objects in order to conclude an existing deadlock in terms of persistent delayed events. This pragmatic, novel notion gives rise an effective deadlock detection algorithm. Therefore, it bridges the gap between deadlock descriptions and deadlock detection algorithms.

The corresponding detection algorithm ensures the liveness property addressed in [JBrzezinski 95, AKshemkalyani 94]. Each detection routine of the algorithm will cease autonomously in a finite time upon either of the three states: a deadlock is revealed, a specified casual relations among objects is no longer maintained, and no further existing casual relation emanated from the current-investigated-object likely leads to a potential deadlock at this moment. Also, this algorithm holds safety property as well; it detects not only the persistent deadlock, existing at the time when a detection routine is initiated, but also a progressive deadlock along its investigating path. The detection on a progressive deadlock is not mentioned in [JBrzezinski 95, AKshemkalyani 94].

6.4.1.1 Deadlock Scenarios

We consider four potential deadlock scenarios depicted in Fig. 6, 7, 8 and 9 in terms of distinct characteristic of activities among objects. A deadlock of scenario I is caused by a circular chain of sequential service requests on distributed objects. A deadlock of scenario II arises from the alternate activities of a sequential invocations and race condition which is considered a typical dinning philosophers problem [KChandy 88]. Upon the presence of joint-action activities, the potential deadlock scenarios III & IV should be considered. A deadlock of scenario III is deemed in regard to that more than one joint-action activity are simultaneously competing for exclusive access to its participant objects. with respect to joint-action activity [DChampeaux 93, Tampere 90], a deadlock of scenario IV is introduced regarding object aliasing [JHogg 92].

In the following Fig. 6, 7, 8, and 9, each arrow indicates the direction of a service request on an object, and a concatenated arrow line indicates a sequential requests, and a vertical dashed line indicates a finite sequence of service request(s) on distinct object(s). A shadowed object indicates that the object itself is locked as it serves a specific incoming request; otherwise an object can be either locked for exclusive access or unlocked. An outward arrow line with multiple inward arrow lines indicates that the outgoing request (i.e., the outward arrow line) is triggered by either of the inward arrow lines. An inward arrow line concatenated with multiple outward arrow lines indicates that the inward request triggers a joint-action activity.
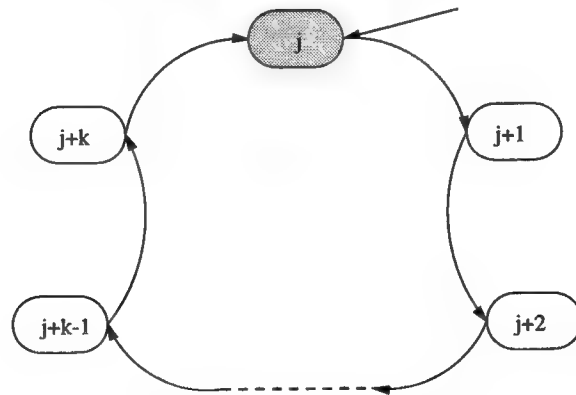
Figure 6: Deadlock Scenario I

$$i_{n-1}+1\leq i_n$$
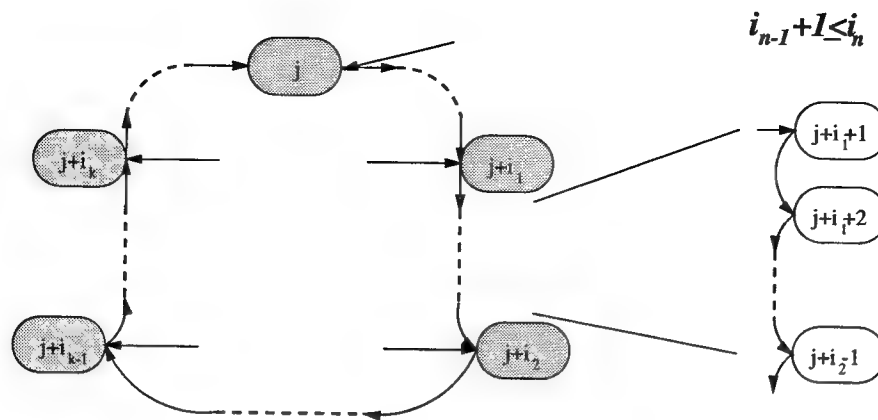
Figure 7: Deadlock Scenario II
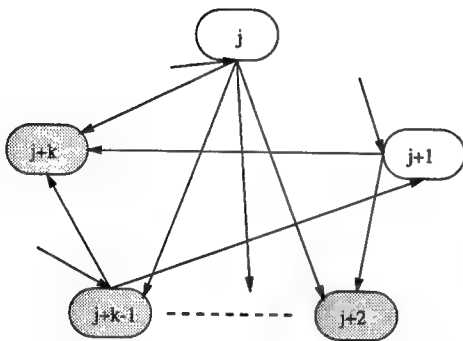
Figure 8: Deadlock Scenario III
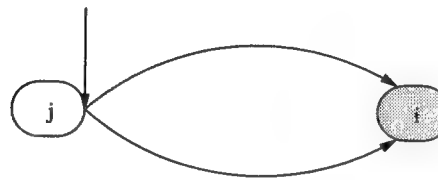
Figure 9: Deadlock Scenario IV

6.4.1.2 Deadlock Descriptions

The novel notion to observing a deadlock, formed by a dynamic group of objects, is a description of the reasoning as to whether the following statement is true: for every object in the group along an investigated path, a reply of a specified service request on its successor object is persistently expected by the local object, where this reply leads to the completion of its corresponding delayed event(s) of local object itself, and this completion eventually fires an associated delayed event of the local object's predecessor.

In order to describe various potential deadlock scenarios in distributed object systems, we first brief on the terminology used here. A context is generated with respect to a service request, such as method invocation or lock request. As an executing context a requests a service, a corresponding context b is generated; the executing context a is a direct parent of the context b. Each generated context naturally inherits the set of sequent history data held

36

by its direct parent context, and it appends a new data of its direct parent context to form its own set of sequent history data. Clearly, a set of history data is accumulated to keep track of a sequence of service requests to the generation of its associated context. Then, we define some notations and assertions:

C_a denotes a context a, O_j denotes an object with a labeled Id of j,

(C_a, O_j) denotes a pair of data, a context a and an object j whereon the context a is generated,

$request_i(C_a, O_j)$ denotes a two-way service request on object i, and the request is issued during the course of executing the task of (C_a, O_j),

$history(C_a, O_j)$ denotes a set of sequent history data held by context a,

$parent(C_a, O_j) = (C_b, O_i) \Rightarrow$
$\qquad C_b$ is a direct parent of $C_a \land request_j(C_b, O_i) = (C_a, O_j) \land$
$\qquad history(parent(C_a, O_j)) - history(C_a, O_j) = \phi \land$
$\qquad history(C_a, O_j) - history(parent(C_a, O_j)) = \{(C_b, O_i)\}$,

$wait\_for\_who(C_a, O_j)$ denotes a set of request(s), in which context C_a on object j is expecting a reply of every request in order to continue the task of (C_a, O_j),

$granting\_participant(C_a, O_j)$ denotes a set of participant objects, in which object j is granted the exclusive access to each participant for the requests issued by context (C_a, O_j),

$locked\_by\_who(O_i) = (C_a, O_j) \Rightarrow$ object i is locked by a context a generated on object j,

$Lock(O_j)$ is an assertion as to whether object j is locked and
$\qquad Lock(O_j) \Rightarrow locked\_by\_who(O_j) \in \{(C_a, O_j), parent(C_a, O_j)\}$,

$Complete(request_i(C_a, O_j))$ is an assertion as to whether an object i has satisfied (or rejected) the request made by (C_a, O_j), either effect leads to complete the task of (C_a, O_j) eventually, and

$Generated(O_i, (C_a, O_j))$ is an assertion as to whether (C_a, O_j) is generated on object i.

In addition, a compound assertion is defined for readability:

$Intra\_object((C_a, O_j), (C_b, O_i))$ is an assertion as to whether a finite sequential request(s) initiated by (C_b, O_i) to the generation of (C_a, O_j) is made on object k only where $k \neq j$;
$\qquad Intra\_object((C_a, O_j), (C_b, O_i)) \Rightarrow$
$\qquad\qquad history(C_b, O_i) - history(C_a, O_j) = \phi \land$

37

$$history(C_a, O_j) - history(C_b, O_i) = \{(x,y)|(x,y), Generated(O_k,(x,y)), k \neq j\}.$$

Then, four potential deadlock scenarios formed by any set D of partial existing objects $k+1$, $k = 1, 2, ...,$ are defined: for any context a, b, d, x, and for any object $ID, c, i, j, k, l, m, n, s, y \in N$,

case 1. $i < j + k \Rightarrow m = i + 1$ and

case 2. $i = j + k \Rightarrow m = j$,

Deadlock scenario I: $\exists\ request_j(C_a, O_i),\ j < i,\ n \leq j + k,\ s.t.$
  $locked\_by\_who(O_j) \in history(request_j(C_a, O_i))\ \wedge$
  $locked\_by\_who(O_j) \neq (C_a, O_i) \wedge$
  $\forall\ O_n\ \exists\ (C_b, O_{n-1})\ [$
    $request_n(C_b, O_{n-1}) \in wait\_for\_who(C_b, O_{n-1}) \wedge$
    $(C_b, O_{n-1}) \in history(request_j(C_a, O_i))\quad]$

Deadlock scenario II: $\forall\ O_i,\ \exists\ (C_a, O_{i-1}),\ \exists\ (C_d, O_m),\ j < i \leq j + k,\ s.t.$
  $\neg Complete(request_i(C_a, O_{i-1})) \wedge$
  $request_i(C_a, O_{i-1}) \in wait\_for\_who(C_a, O_{i-1}) \wedge$
  $(\neg Lock(O_{i-1})\ \vee\ locked\_by\_who(O_{i-1}) \in history(request_i(C_a, O_{i-1}))\ ) \wedge$
  $(\ (locked\_by\_who(O_i) \notin \{(C_a, O_{i-1}), request_i(C_a, O_{i-1})\}\ \wedge$
    $locked\_by\_who(O_i) \in history(C_d, O_m) \wedge$
    $Intra\_object((C_d, O_m), locked\_by\_who(O_i))$
    $)\ \vee$
  $(\ (\neg Lock(O_i) \vee locked\_by\_who(O_i) \in \{(C_a, O_{i-1}),\ request_i(C_a, O_{i-1})\}\ )\ \wedge$
    $Intra\_object((C_d, O_m), request_i(C_a, O_{i-1}))$
    $)$
  $)\ \wedge \neg Complete(request_m(parent(C_d, O_m)))\ \wedge\ Lock(O_j)$

Deadlock scenario III:
  $C \cup R = D, where\ j \leq c, r, s, n \leq j + k,\ s \neq n,$
  $C = \{O_c|O_c \in D \wedge wait\_for\_who(C_a, O_c) \neq \phi\ \wedge$
        $granting\_participant(C_a, O_c) \neq \phi$
      $\}, and$
  $R = \{O_r|O_r \in \{O_y|request_y(C_d, O_s) \in wait\_for\_who(C_d, O_s)\} \cap$
          $granting\_participant(C_b, O_n)\ \wedge$
        $\forall O_s, O_n \in C\ \wedge\ locked\_by\_who(O_r) = (C_b, O_n)\ \wedge$
        $\neg Complete(request_r(C_d, O_s))$

```
    }
```

Deadlock scenario IV: $\exists\ O_i,\ \exists\ O_l,\ j < i,\ l \le j + k, s.t.,$

$\quad \neg Complete(request_l(C_b, O_j))\ \wedge$

$\quad request_l(C_b, O_j) \in wait\_for\_who(C_b, O_j)\ \wedge$

$\quad O_i \in granting\_participant(C_b, O_j)\ \wedge$

$\quad locked\_by\_who(O_i) = (C_b, O_j)\ \wedge\ l = i$

6.4.1.3 Detection Algorithm

The deadlock cycles of persistent-delayed-events in any mixed activities of the four above-mentioned deadlock scenarios can form in innumerable ways. Therefore, a detection algorithm should hold the safety property, capable of reasoning any persistent deadlock.

A deadlock detection routine is initiated by any O_i as a runnable context of $request_i(C_a, O_j)$ is deactivated because the true condition described in the following statement:

$Lock(O_i)\ \wedge\ locked\_by\_who(O_i)\ \notin\ \{request_i(C_a, O_j), (C_a, O_j)\}.$

Later, this autonomous routine will either cease on detecting a persistent deadlock along its investigated path, or cease on first encountering a breakage of a potential deadlock. The breakage arises from that, either the evaluation that a current-visited-object i'$wait\_for\_who(locked\_by\_w$ $\phi\ \vee\ wait\_for\_who(request_i(C_a, O_j)) = \phi$ is true, or the object i has completed (ie., satisfied or rejected) a specified request issued by C_a of its predecessor object j (ie., (C_a, O_j)) and the visited object i itself is no longer engaged in a corresponding activity issued by (C_a, O_j).

As a detection routine is initiated, it first examines locally as to whether a deadlock of scenario I exists. If not, concurrent depth-first investigations will be applied to determine whether persistent deadlock(s) exists. The number of concurrent depth-first investigations is conditioned corresponding to the length of $wait\_for\_who(locked\_by\_who(O_i))$. Each investigation will cease on the state of the above-mentioned breakage; otherwise, a next group of candidate objects (ie., successors) to be investigated is determined corresponding to either the element(s) of $wait\_for\_who(request_i(C_a, O_j))$, or the element(s) of $wait\_for\_who(locked\_by\_who(O_i))$. Provided a deadlock is detected, the detection routine sends every involving object an inquiry as to whether its associated observed-delayed-event is aborted due to the occurrence of a deadlock resolution. Each reply of the inquiries is collected in order to determine if a corresponding deadlock resolution should be performed. Provided that every observed-delayed-event is delayed persistently, a corresponding deadlock resolution is then performed; thus, a false deadlock is definitely excluded.

Deadlock Resolution Deadlock resolution is very expensive due to the corresponding rollback management. Yet, every deadlock of scenario I is not resolvable. So as to minimize the cost of rollback management an involving joint-action activity is forcibly aborted to break the persistence of deadlock events, while the joint-action event is in the phase of acquiring the exclusive accesses to its participants. .

Optimization on Implementation As for implementing this novel notion to deadlock detections, any object j is constructed with two additional attributes holding the information of $locked\_by\_who(O_j)$ and $wait\_for\_who(C_a, O_j)$ respectively, and any context a is generated with an additional attribute holding the information of $history(C_a)$. To gain better performance, the history data of a context is conditionally constructed provided that the history of its direct parent context exists; otherwise a context is generated with a null history. while being granted exclusive access to its residence object, a null-history context will give out the data of the context itself as a history information to its every direct child context.

6.4.2 Deadlock Avoidance

In this reflective model two dedicated protocols are devised, so as to autonomously avoid the expense of initiating a deadlock detection routine, in order to prevent a deadlock scenario III (see Fig. 8) and resolve a deadlock scenario IV (see Fig. 9), respectively.

Race Condition of Two Objects with Joint-Action Activities If two objects, i and j, send out a joint-action engagement request to each other at the same time, then they lock up. Each object can avoid this type of deadlock proneness by determining if its own current joint-action activity should proceed; the two objects compare their ID with others. The object with higher ID value continues its joint-action activity; consequently, the other one postpones its joint-action activity by issuing a self-assertive notification of "ABORT" to all the other participants. The postponed activity will be resumed after the object releases itself from the joint-action engagement.

Object Aliasing Consider that, in a joint-action guard, more than one atomic guard with distinct identifier actually refer to same object. Deadlock occurs when object j pursues the joint-action guard evaluation. After issuing the joint-action guard evaluation, object j is collecting every reply of being granted exclusive access to each participant object in order to proceed its activity. However, a participant object i with aliasing

is not able to completely satisfy all the engagement requests because the aliasing object i is waiting for a notification first to release itself from the current engagement in order to serve any of queued outstanding requests. For hacking this type of deadlock, a resolution mechanism is imposed so that an object is capable of determining if an incoming engagement request is issued for same joint-action activity in which the recipient object is engaging, and, if it is, then the request should be served.

6.5   Solution for Inheritance Anomaly

The conflict between the inheritance and synchronization constraints breaks the encapsulation feature of object-oriented language and could possibly nullify the benefit of inheritance. Matsuoka and Yonezawa [SMatsuoka 93] propose the total separation of the synchronization code from the guarded method code, thus, it separates the guarded method to be inherited and overridden separately from the synchronization code. Frolund [SFrolund 92] proposes the restriction notion in which, provided the restriction condition, the incoming request will not be serviced and a method of subclass naturally inherits the restriction constraints on the method of the same name in the superclasses. Restriction constraints are accumulated along the inheritance hierarchy. Frolund's work draws attention to the issue of encapsulation for the consistent reuse of a guarded method code. Ferenczi [SFerenczi 95] demonstrates that the nested guarded method call solves the inheritance anomaly.

In this work, in order to target the reuse of a guarded method code, a coding scheme of nest guard is suggested to solve the inheritance anomaly. The coding scheme is demonstrated by using Examples 3.0–3.2. Moreover, a selective aggregation mechanism with in-lining technique is proposed to effectively and efficiently support the coding scheme.

```
class Base                 class Next₁ is Base...         class Next₂ is Next₁ ...

...                        ... // Next₁ inherits Base     ... // Next₂ inherits Next₁ ...

op m₀(...)                 op m₁(...)                     op m₂(...)
  {                          evaluateExclusion(self);       transitExclusion(self);
                             when c₁₁ then m₀(...)           when c₂₁ then m₁(...)
  ...                        else pend end                  else pend end
  }
end                        end                            end
```

| Example 3.0 | Example 3.1 | Example 3.2 |

6.5.1 Coding Scheme

A guarded method code is defined in a superclass, and then a separate method of subclass is defined to describe the added-in guard code. In order to add more guard to a method code, a method of another tier subclass is defined to describe the to-be-added-in guard. So as to override a guard of a specific tier of subclass, the method describing the guard will be redefined to describe a new guard of the corresponding tier of subclass.

6.5.2 Selective Aggregation Mechanism

During compilation time, static analysis *rule 1.a* is applied to determine if the aggregation mechanism using *rule 1.b* should be pursued.

rule 1.a A caller method of subclass invokes a method of superclass; a guard is embedded in the caller method.

rule 1.b The outermost level of the embedded guard of the invoked method of superclass and the innermost level of the embedded guard of its caller method of subclass are aggregated into a form of composite guard with a logical conjunction. With respect to the aggregation, the rest of the invoked method code of superclass is inlined to its caller method during compilation time.

Consider that the method m_2 of class $Next_2$ in Example 3.2 invokes the method m_1 of superclass $Next_1$ in Example 3.1. After the composition of guards and the inlining of method m_1, the complete code of method m_2 of class $Next_2$ is shown in Example 4.2. With regard to the aggregation, the corresponding Exclusion facility of the composite guard is determined by the most intensive atomicity control over an object. The intensity of atomicity control is defined in terms of the lifetime of atomicity control. Therefore, the transitExclusion(*self*) of Example 3.2 is chosen over the evaluateExclusion(*self*) of Example 3.1.

With the selective aggregation mechanism a guarded method code can be reused later as well as overridden. As a matter of fact, the selective aggregation mechanism annihilates the runtime overhead of an intra-object method invocation by inlining an invoked method code into its caller method of subclass.

```
class Next₁ is Base ...                  class Next₂ is Next₁ ...

...                                      ...
op m₁(...)                               op m₂(...)
                                           transitExclusion(self); ⇐guard conjunction and
  evaluateExclusion(self);                 when c₂₁ and c₁₁ then    inlining m₁(...) of class Next₁
                                               { ⇐ inlined m₀() of class Base
  when c₁₁ then                                ...
      { ⇐ inlining m₀() of class Base          }
                                           else pend end
      ...                                  end
      }
```

$$class\ Next_1\ is\ Base\ ...$$
$$op\ m_1(...)$$
$$evaluateExclusion(self);$$
$$when\ c_{11}\ then$$
$$\{\ \Leftarrow inlining\ m_0()\ of\ class\ Base$$
$$...$$
$$\}$$
$$else\ pend\ end$$
$$end \qquad Example\ 4.1$$

$$class\ Next_2\ is\ Next_1\ ...$$
$$op\ m_2(...)$$
$$transitExclusion(self);\ \Leftarrow\textbf{guard conjunction}\ and$$
$$when\ c_{21}\ and\ c_{11}\ then \quad \textbf{inlining}\ m_1(...)\ of\ class\ Next_1$$
$$\{\ \Leftarrow inlined\ m_0()\ of\ class\ Base$$
$$...$$
$$\}$$
$$else\ pend\ end$$
$$end$$

Example 4.2

6.6 Why Use an Event-Driven Approach?

While designing a synchronization model to implement the ODL guard method, both polling
and event-driven approaches have been considered. Generally speaking, a polling approach is
simple, yet undesirable for two reasons. First, it takes up some portion of CPU time for the
busy-waiting (i.e., polling) on a specified guard before its corresponding pended transition(s)
is fired. Second, in the case of periodically evaluating a joint-action guard, every polling on
the requested state(s) of each participant object also increases the underlying communica-
tion network traffic. Besides, for ensuring consistent post-condition of a joint-action guard
evaluation, all the participants have to remain atomic during the guard evaluation. The
atomicity control inhibits the potential concurrency, and consequently reduces the opportu-
nity for overlapping computation well with the communication among objects to gain better
performance. Precisely speaking, an over-frequent polling approach significantly degrades
the application performance. However, a less frequent one will not be able to meet the
demand of a real-time reactive application system. Therefore, we are in favor of an event-
driven approach. Through the adoption of an event-driven approach, a guard will only be
reevaluated upon the change of the corresponding state(s) of an object.

6.7 The Optimized Event-Driven Reflective Model

The event-driven reflective model is devised with every possible aspect of optimizations. To
target all the optimizations, the static analysis rules "2–5" are devised regarding both the
minimization on the runtime overhead of event-driven activity and the reductions on the
frequency of context switching, the communication latency, and message process latency.
Along with the optimization on the reductions, the reflective model achieves the concurrent

43

evaluation on a joint-action guard by transforming synchronous method invocations into asynchronous invocations. In this event-driven reflective model some runtime components are designed with a fully decentralized management to coordinate a group of participant objects for joint-action activity.

6.7.1 Static Analysis Techniques

Two major static analysis tasks are devised for supporting the incremental inter-method dependency approach and message grouping mechanism.

6.7.1.1 The Incremental Inter-method Dependency Approach

Due to the dynamic behavior of a method at runtime, a method code could possess several potential terminate points. Each terminate point might reflect a distinct set of all the methods depending on a method defined within the same class. Clearly, a unique inter-method dependency table for every terminate point of a method is not efficient enough to support event-driven activity. Therefore, the incremental inter-method dependency approach is devised.

To support the incremental inter-method dependency approach, each potential terminate point of a method should be determined by using *rule 4*, which is devised according to the EBNF syntax of the ODL language. Over the intermediate SIRF code generation stage every corresponding inter-method dependency table for each potential terminate point of a method code is generated by using its associated set of dependency data collected by using *rule 2* and *3*; *rule 3* analyzes the inter-method dependency among any two methods within same ODL class, and *rule 3* analyzes an indirect inter-method dependency. The application of *rules 2* and *3* is demonstrated in Example 5. As a matter of fact, each inter-method dependency table is minimized as much as possible to reduce the lookup for pended contexts, as well as to reduce the scheduling and context-switching for the reevaluation of some pended contexts.

Rule 2 Within an ODL class a method A is a dependent of a method B if and only if the intersection of USE(A) and DEFINE(B) is nonempty, where each element of the USE and DEFINE sets represents a class-scope state variable. USE(A) presents a set of class-scope state variables that is used potentially over the course of execution of method A. The DEFINE(B) presents a set of class-scope state variables that is defined

potentially over the course of execution of method B.

Rule 3 Consider that a method A is of an atomic guard. Considered as an event-driven activity, any guard containing method A should be reevaluated upon the state change of method A. Therefore, any method enclosing a guard that contains method A will be a dependent of every method dominating the state of method A; the corresponding dependency data will be collected.

```
class B
a : int;
b : int;
op op1 : bool
  {
     ⋮
  c := a + ...
     ⋮
  }
op op2(Num : int)
  when op1 and b > Num then
  {
     ⋮
  }
  else pend end

op op3(maxVal : int) : int
  {
  while(a < maxVal) do
     ⋮
  a := ...
     ⋮
  end
     ⋮
  }
end;
```

Example 5

```
op Sample{
     ⋮
  statements
     ⋮
  if exp11 then
   if exp121 then
     ...statements...
   elseif exp122 then
     ...statements...
   else ...statements ...end
     ⋮                        ⇐ aggregation point
  statements
     ⋮                        ⇐ terminate point 1
  elseif exp12 then
     ...statements...
   if exp221 then
     ⋮
  statements
     ⋮                        ⇐ terminate point 2
  elseif exp222 then
     ⋮
  statements
     ⋮                        ⇐ terminate point 3
  else ...statements...       ⇐ terminate point 4
   end
  elseif exp13 then
     ⋮
  statements
  end                         ⇐ terminate point 5
}
```

Example 6

In Example 5 an ODL class B is defined as being composed of two class-scope states, a and b, and three method codes, *op1*, *op2*, and *op3*. It is observed by applying *rule 2* that method *op2* is a dependent of method *op1* and method *op1* is a dependent of method *op3*. In addition, once *op3* is invoked and executed, the state of method *op1* is changed. Thus, by applying *rule 3*, *op2* is a dependent of *op3*.

45

Rule 4. According to the ODL's EBNF syntax, a method code containing at least one statement with conditional jump(s), such as the if, when, and while statements, might have distinct terminate points at runtime. A "terminate point" is defined with regard to the end of a block of statements corresponding to a conditional jump of a last statement in a method code. "Aggregation point" is the term used when a statement with conditional jump(s) is not the last statement of a method code being parsed—that is, where all the individual sets of collected dependency data for each conditional jump are aggregated to form a whole set of dependency data. The aggregation takes place after the next statement is parsed. In accordance with the aggregation, the corresponding inter-method dependency set most likely grows. For instance, a framework of a method Sample, demonstrated in Example 6, has five potential terminate points and one aggregation point.

6.7.1.2 The Message-Grouping Mechanism

Static analysis *rule 5* is devised to support the message- grouping mechanism in order to reduce both communication latency and message process latency so as to target further optimization on the concurrent evaluation on a joint-action guard.

Rule 5. Every object ID addressed within each atomic guard is analyzed, and all the atomic guards addressed to the same object are grouped into a unit of group request.

At runtime a unit of group request on an object is transformed into a ReflectionMessage format and is transported as a unit of a message to its recipient object. This approach reduces the multiple communication latencies of multiple requests into only one communication latency, and it reduces the multiple latencies of multiple arriving requests waiting on the to-be-served-queue at the recipient object site into only one latency of waiting-on-the-queue-to-be-served. Besides, the message-grouping mechanism significantly reduces the portion of time for marshalling and unmarshalling both the requests of method invocations and the returned replies of the requests. The reduction is achieved by marshalling the multiple requests into a unit of group requests and by unmarshalling the group requests only once from the underlying distributed computing environment.

6.7.2 Major Components for Runtime Management

The components (see Fig. 10) are designed to implement the optimized event-driven reflective model. The functionality of the major components is briefly described in the following.
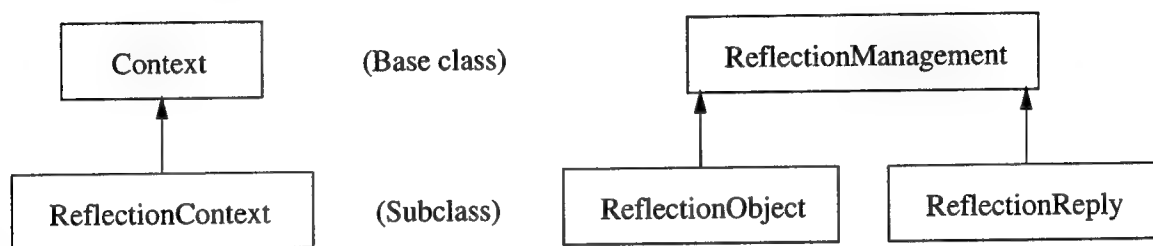

Figure 10: Major Runtime Components

Transforming Synchronous Invocation into Asynchronous Invocation The class ReflectionContext is designed to support the transformation of synchronous invocation into asynchronous invocation in order to reduce the frequency of context switching into exactly one. In general, an instance of Context is constructed upon the reception of a method invocation request. However, each instance of ReflectionContext is constructed for each method invocation request embedded in a "JointAction" engagement request for a joint-action guard evaluation activity.

Synchronizing Joint-Action Activities The C++ classes, ReflectionManagement, ReflectionObject, and ReflectionReply, are designed to implement the coordination task for joint-action activity among a dynamic group of distributed participant objects. A ReflectionObject is responsible for collecting all the replies of requested state(s) from each participant instance of ReflectionReply(s) in order to proceed with a corresponding guard evaluation. Therefore, this design approach of the three runtime components gives rise to annihilating the cost of generating every instance of a specialized class for receiving and disposing a returned reply regarding each request on every state of a joint-action guard within a joint-action activity.

An instance of ReflectionObject is constructed at the very first time when a SIRF code of "JointActionGuard" is executed. Following the construction, the executing context issues a sequence of "JointAction" engagement requests to its participant objects. In each request the associated method invocation(s) for each sub-guard is enclosed. Upon the reception of a "JointAction" engagement request each other participant object constructs an instance of ReflectionReply with regard to a joint-action activity at the
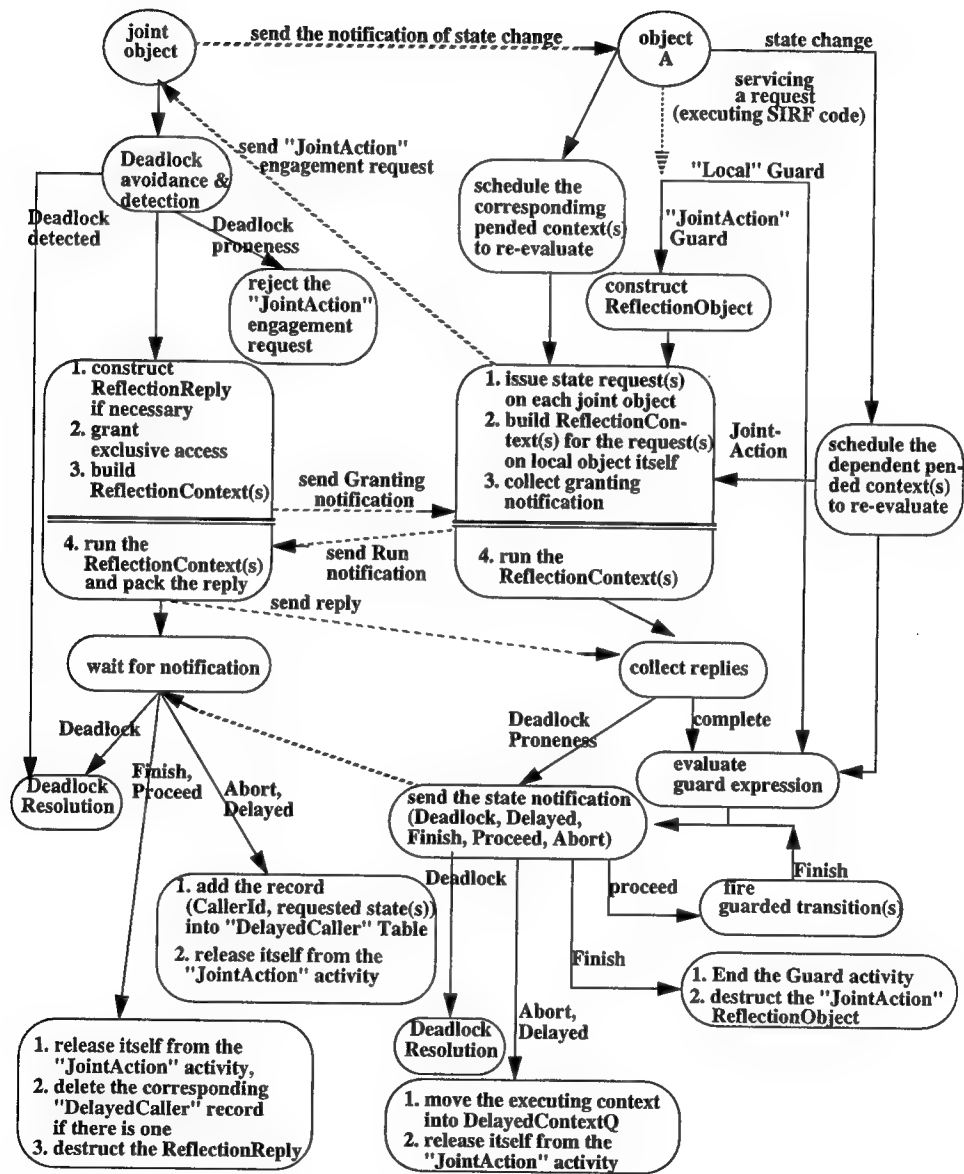
47

Figure 11: The Block Diagram of Event-Driven Activity of the ODL Guard Method

very first request. Each constructed instance will not be destructed until the completion of the corresponding joint-action activity. The exchange of notification for joint-action activity is encapsulated in the ReflectionMessage format. With this design notion, a hierarchical state table is constructed for processing an incoming message from the underlying network. In this way it does not affect the performance of the existing Diamonds computing framework as this reflective model is integrated.

The other seven distinct notifications are defined for synchronizing the activities among distributed objects. A "Granting" notification is given by a participant object committing to the engagement in a joint-action activity. After collecting every "Granting" notification from each participant, the recipient object of the executing context issues a "Run" notification to each participant objects to process the corresponding request on its state(s) that is in part of joint-action guard. It then collects each reply of every participant for the evaluation on the joint-action guard. If the joint-action guard is evaluated to be true, the executing context will send a "Proceed" notification to inform each participant object of its firing on the corresponding guarded transition(s). When the guarded transition(s) is completed, a "Finish" notification will be sent to each participant; otherwise, a "Delayed" notification will be sent. A self-assertive notification of "Abort" will be issued only when the object of the executing context detects its corresponding joint-action activity to be deadlock-prone. For instance, if any two objects, i and j, send out a "JointAction" engagement request to each other at the same time, they lock up. Each of the two objects can avoid deadlock proneness by autonomously determining if its current joint-action activity should proceed; the two objects compare their own ID with others. The object with a higher ID value continues to proceed with its joint-action activity; consequently, the other one postpones its own joint-action activity by issuing an "Abort" notification to all the other participants. The postponed activity will be resumed after the object releases itself from the other engagement of joint-action activity. As a persistent circular chain of deadlocked events is revealed, a notification of "Deadlock" is issued. A block diagram of event-driven activity of the ODL guard method is depicted in Fig. 11.

Extending Each Object With Three Queues In order to support event-driven activity each ODL object is constructed with three distinct queues: WaitOnLockQ, DelayedContextQ, and DelayedCallerQ. The WaitOnLockQ is a repository for queuing a scheduled context(s) that is expecting the event where its residence object is expected to be released (i.e., unlocked). As the SIRF code of "Pend" is executed, the current

active context is swapped out and moved to the DelayedContextQ. As a joint-action activity is delayed at the very first time, apart from the caller object issuing the engagement request(s), each other participant object records the delayed activity with the record of the caller ID and the requested method's ID into its DelayedCallerQ. Upon the change of any requested object's state, all the pending events depending on the change of the specific state(s) will be resurrected and scheduled to run.

6.8 Effectiveness of the Optimized Reflective model

In this subsection, we formulate a measurement of concurrent evaluation vs. serial evaluation on the ODL joint-action guard to show the effectiveness of the optimized reflective model. In the measurement, some factors are taken into account and described below. Let

$request_i$ denote a request i, $reply_i$ denote the reply of a request i,

T_{total} denote the total time needed to access every state of each sub-guard expression,

$T_{exec}(reguest_i)$ denote the execution time of method i,

$T_{context-switch}$ denote the time needed for context switching,

$L_{queue}(j)$ denote the latency to wait on the ready-to-run-queue on object j,

L_{convey} denote the average latency to convey a message from an object to another remote object,

$T_{protocol}(i)$ denote the time needed to marshal or unmarshal either a $request_i$ or a $reply_i$ at the level of underlying networking facilities (i.e., TCP/IP, Socket...), and

$T_{diamonds}(i)$ denote the time needed either to marshal a request i or $reply_i$, which will probably be conveyed to the other cluster, or to unmarshal an incoming message from another cluster at the level of the Diamonds computing environment.

The infrastructure of the Diamonds computing environment [KShank 94] is designed to achieve maximum concurrency of each object-oriented application system running on it. Therefore, objects with a higher frequency of interactions are collocated into the same cluster to avoid the latency of inter-cluster communication. In the Diamonds computing environment it takes time $T_{inter-cluster}$ for an object A to issue a request of method i on a remote

object B and to receive a reply of the request; $T_{inter-cluster}$ is described in Equation (1).

$$T = 2(T_{diamonds}(request_i) + T_{protocol}(request_i) + L_{queue}(B) +$$
$$T_{exec}(request_i) + 2(T_{protocol}(reply_i) + T_{diamonds}(reply_i))). \qquad (1)$$

A returning reply is privileged to be processed upon its arrival at its designated object site. Therefore, factor $L_{queue}(A)$ is not taken into account in Equation (1). In the Diamonds computing environment, an inter-object method invocation triggers either inter-cluster communication or intra-cluster communication. Equation (1) is used to measure the throughput $T_{inter-cluster}$ of the inter-object method invocation triggering an inter-cluster communication. The cost of the inter-object method invocation within a cluster is very much optimized to that of the intra-object method invocation. The throughput $T_{intra-cluster}$ for an inter-object method invocation triggering intra-cluster communication is measured by Equation (2).

$$T_{intra-cluster} = L_{queue}(B) + T_{exec}(request_i). \qquad (2)$$

In addition, consider that an intra-object method invocation is not recognized at compilation time. This type of invocation happens because of object aliasing (i.e., the designated object B of the invoked method is resolved to be an aliasing of the object A issuing the request for the method invocation). Then, the throughput $T_{aliasing}$ of the invocation on method i is measured by Equation (3).

$$T_{aliasing} = T_{diamonds}(request_i) + L_{queue}(B) + T_{exec}(request_i). \qquad (3)$$

The effectiveness of the devised optimized reflective model is demonstrated by using the joint-action guard activity of Example 2, in which there are three objects, a, b, and *local* object itself, engaging in a joint-action guard activity. In this joint-action engagement the

```
transitExclusion(a, b); evaluateExclusion(self);
when(a.M_a(...) + b.M_{b1}(...) + b.M_{b2}(...) + Local(...)) > 0
then
 { ... do something; }
else pend end
```

Example 2.

state of method M_a on remote object a, both the states of method M_{b1} and M_{b2} on remote object b, and the state of method *Local* on local object itself are requested in order to

51

evaluate the joint-action guard expression. Assume that the underlying network traffic is not jammed. Consider that along the course of executing the task of Example 2 a local object encounters the SIRF code of the joint-action guard at a point of time t_0. Some parameters are inductively described below. Let

t_1 denote the time when the request of M_a on object a is ready to be sent to underlying network,

t_2 denote the time when the requests of M_{b1} and M_{b2} on object b are ready to be pumped into underlying network channel,

t_3 denote the time when object a site receives the request of M_a,

t_4 denote the time when object b site receives the requests of M_{b1} and M_{b2},

T_{Local} denote the time when the local object completes the execution on the method $Local$,

T_a denote the time when the local object receives the reply result from remote object a, and

T_b denote the time when the local object receives the reply results from remote object b.

Then

$$t_1 = t_0 + T_{diamonds}(M_a) + T_{protocol}(M_a),$$

$$t_2 = t_1 + T_{diamonds}(M_{b1} + M_{b2}) + T_{protocol}(M_{b1} + M_{b2}),$$

$$t_3 = t_1 + L_{convey}, \qquad t_4 = t_2 + L_{convey},$$

$$T_{Local} = t_2 + T_{context-switch} + T_{exec}(Local),$$

$$T_a = t_3 + T_{protocol}(M_a) + T_{diamonds}(M_a) + T_{Lqueue}(a) + T_{exec}(M_a) + \\ T_{diamonds}(reply_{M_a}) + T_{protocol}(reply_{M_a}) + L_{convey},$$

$$T_b = t_4 + T_{protocol}(M_{b1} + M_{b2}) + L_{queue}(b) + T_{exec}(M_{b1} + M_{b2}) + \\ T_{diamonds}(reply_{(M_{b1}+M_{b2})}) + T_{protocol}(reply_{(M_{b1}+M_{b2})}) + L_{convey}.$$

With the proposed optimized model the measurements for the throughput T_{total} of concurrent evaluation on all the sub-guards stated in Example 2 are formulated in Equations (4)–(7), where $Max(x_1, x_2, ..., x_n) = x_i \Rightarrow x_i > x_j, \forall i, j \in N$ and $i \neq j$. The Equations (4)–(7) are used to measure three distinct communication scenarios that are described below, respectively. To show the effectiveness of the optimized model, the throughput T_{serial} of serial evaluation on all the sub-guard expressions is measured as well, using Equation (8). Two

notations, $T_{message}(i)$ and $T_{turnaround}(i)$, are defined in order to improve the readability of Equation (8). $T_{message}(i)$ denotes the time needed to marshal and unmarshal both a request of method i invocation on a remote object and a reply of the request at both levels of the Diamonds computing framework and its underlying networking facilities. $T_{turnaround}(i)$ denotes the time needed to access the state of a method i.

Optimized Concurrent Evaluation:

Scenario 1.

If $Max(T_{Local}, T_a, T_b) = T_{Local}$, then

$$
\begin{aligned}
T_{total} &= T_{Local} + T_{protocol}(reply_{M_a}) + T_{diamonds}(reply_{M_a}) + \\
&\quad T_{protocol}(reply_{M_{b1}+M_{b2}}) + T_{diamonds}(reply_{M_{b1}+M_{b2}}).
\end{aligned} \tag{4}
$$

$$\tag{5}$$

Scenario 2.

If $Max(T_{Local}, T_a, T_b) = T_a$, then

$$
\begin{aligned}
T_{total} &= T_{protocol}(reply_{M_a}) + T_{diamonds}(reply_{M_a}) + \\
&\quad Max(T_a, Max(T_{Local}, T_b) + \\
&\quad T_{protocol}(reply_{M_{b1}+M_{b2}}) + T_{diamonds}(reply_{M_{b1}+M_{b2}})).
\end{aligned} \tag{6}
$$

Scenario 3.

If $Max(T_{Local}, T_a, T_b) = T_b$, then

$$
\begin{aligned}
T_{total} &= T_{protocol}(reply_{M_{b1}+M_{b2}}) + T_{diamonds}(reply_{M_{b1}+M_{b2}}) + \\
&\quad Max(T_b, Max(T_{Local}, T_a) + T_{protocol}(reply_{M_a}) + T_{diamonds}(reply_{M_a})).
\end{aligned} \tag{7}
$$

Serial Evaluation:

$$T_{serial} = T_{turnaround}(Local) + T_{trunaround}(M_a) + T_{turnaround}(M_{b1}) + T_{turnaround}(M_{b2}). \tag{8}$$

where

$$T_{message}(i) = 2(T_{diamonds}(i) + T_{protocol}(i)) + 2(T_{diamonds}(reply_i) + T_{protocol}(reply_i)),$$
$$T_{turnaround}(Local) = T_{context-switch} + L_{queue}(self) + T_{exec}(Local),$$
$$T_{turnaround}(M_a) = T_{context-switch} + L_{queue}(a) + T_{exec}(M_a) + T_{message}(M_a),$$
$$T_{turnaround}(M_{b1}) = T_{context-switch} + L_{queue}(b) + T_{exec}(M_{b1}) + T_{message}(M_{b1}),$$
$$T_{turnaround}(M_{b2}) = T_{context-switch} + L_{queue}(b) + T_{exec}(M_{b2}) + T_{message}(M_{b2}).$$

Let $O_{joint-action}$ denote a set of participant objects engaging in a joint-action guard evaluation and let $|O_{joint-action}|$ denote the number of participant objects. $|O_i|$ denotes the numbers of the state(s) of a participant object i, and each state is in part of a joint-action guard. By taking into accounts the Equations (2) and (3) and comparing each T_{total} of Equations (4)–(7) with the T_{serial} of Equation (8), Equation (9) is derivatively formulated to describe the measurement of the optimized concurrent evaluation versus serial evaluation on a joint-action guard of k-number states.

Optimized Concurrent Evaluation vs. Serial Evaluation :

$$T_{serial} - T_{total} \geq (k-1) * T_{context-switch} + \sum_{i=1}^{|O_{joint-action}|} (L_{queue}(O_i) * (|O_i| - 1)). \qquad (9)$$

6.9 Conclusion

In this work, the optimized reflective model devised for implementing the ODL guard method is very much optimized in every aspect of optimization technique. We are convinced that our design of run-time components with a complete decentralized management yields an effective and efficient computing model for coordinating a dynamic group of distributed objects.

Both incremental inter-method dependency and message grouping approaches are proposed with their corresponding static analysis rules, which are devised in order to minimize the runtime overheads of the associated approaches. The unique incremental inter-method dependency approach gives rise to every distinct set of all the dependent methods corresponding to each potential terminate point of each method within the same class. In other words, the information of the intra-class inter-method dependency is statically captured regarding a potential dynamic behavior of a method. With this technique to support event-driven activity, the runtime overhead of collecting the dynamic information of inter-method dependent set can be alleviated. We believe that these notions of the devised static analysis rules

54

are of great value in development research on high-performance concurrent programming languages.

In addition, a novel description of deadlock reasoning is proposed in this work for concluding a deadlock based upon only a minimum set of persistent temporal information of casual relations among a dynamic group of objects which form the deadlock. This description yields a very effective deadlock detection algorithm. This algorithm retains scalability as a system evolves to grow, in terms of the number of existing objects. Most of all, this notion of deadlock descriptions is a very first description with a pragmatic approach to attack the deadlock detection problem in development research on open, distributed system environment.

Bibliography

[ATK92]   A. L. Ananda, B. H. Tay, and E. K. Koh. A Survey of Asynchronous Remote Procedure Calls. *Operating Systems Review*, 26(2):92–109, April 1992.

[BCLS93]   Umesh Bellur, Gary Craig, Doug Lea, and Charles K. Shank. Clusters: A Pragmatic Approach Towards Supporting a Fine-Grained Active Object Model in Distributed Systems. In *9th International Conference on Systems Engineering*, Las Vegas, Nevada, July 1993.

[BCSL93]   Umesh Bellur, Gary Craig, Kevin Shank, and Doug Lea. DIAMONDS: Principles and Philosophy. Technical Report 9313, CASE Center, Syracuse University, June 1993.

[Bir85]   A. Birrell. Secure Communication Using the Remote Procedure Call. *ACM Transactions on Computer Systems*, 3(1), 1985.

[Bir93]   K. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):36–53, December 1993.

[Boo94]   G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, 2nd edition, 1994.

[CBHS93]   Vinny Cahill, Shean Baker, Chris Horn, and Gradimir Starovic. The Amadeaus GRT - Generic Runtime Support for Distributed Persistent Programming. In *OOPSLA '93*, pages 144–161, 1993.

[Cea94]   D. Coleman and et. al. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, 1994.

[CGZ94]   Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying Behavioral Differences Between C and C++ Programs. Technical Report CU-CS-698-94, Department of Computer Science, University of Colorado, 1994.

[CKP93]   Andrew Chien, Vijay Karamcheti, and John Plevyak. The Concert System –
          Compiler and Runtime Support for Efficient, Fine-Grained Concurrent Object-
          Oriented Programs . DCS Technical Report UIUCDCS-R-93-1815, University of
          Illinois, Department of Computer Science, 1304 W. Springfield Avenue, Urbana,
          Illinois, June 1993.

[CLNZ89] S. K. Chung, E. D. Lazowska, D. Notkin, and John Zahorjan. Performance
          implications of design alternatives for remote procedure call stubs. In *The 9th In-
          ternational Conference on Distributed Computing Systems*, pages 36–41, Newport
          Beach, CA USA, June 1989. IEEE.

[Cor90]   IBM Corporation. *POWER Processor Architecture.* Advanced Workstation Di-
          vision, Austin, TX, 1990.

[CR92]   Arunodaya Chatterjee and William A. Rogers. Integrated Resouce Allocation in
          ESP. Technical Report ESL-ESP-194-92, MCC, October 1992.

[Cra94]   G. Craig. Resource Management with Application Information Exchange. In
          IEEE Dual-Use Technologies and Appl. Conf., 1994. available as dual-use94.ps.Z
          on leopard.cat.syr.edu.

[CUL89]   Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of
          SELF – a dynamically-typed object-oriented language based on prototypes. In
          Proceedings OOPSLA '89, pages 49–70, October 1989. Published as ACM SIG-
          PLAN Notices, volume 24, number 10.

[dCLF93] Dennis de Champeaux, Doug Lea, and Penelope Faure. *Object-Oriented Software
          Development.* Addison-Wesley, 1993.

[Dic91]   Peter Dickman. Effective Load Balancing in a Distributed Object-Support Op-
          erating System. In *1991 Workshop on Object Orientation in Operating Systems*,
          1991.

[DLea 94] D. Lea and K. Shank, *ODL: language report,* CASE center at Syracuse Univ.,
          Dec. 1994

[Dra90]   Richard P. Draves. The Revised IPC Interface. In *Proceedings of the USENIX
          Mach Conference*, October 1990.

[Gol83]   Adele Goldberg. *Smalltalk 80: The Language and its Implementation*. Addison-Wesley, 1983.

[HJ86]   Anna Hac and Theodore J. Johnson. A Study of Dynamic Load Balancing in a Distributed System. In *SIGCOMM Sym. on Communications Architectures and Protocols*, pages 348–356, August 1986.

[HK87]   C. Horn and S. Krakowiak. Object Oriented Architecture for Distributed Office Systems. In *Procedings of Esprit Conference*, 1987.

[Inc87]   Sun Microsystems Inc. *The SPARC Architecture Manual*. Mountain View, CA, 1987.

[Jea92]   I. Jacobson and et. al. *Object-Oriented Software Engineering*. Addison-Wesley, 1992.

[JKS91]   H.-M. Jarvinen and R. Kurki-Suonio. DisCo Specification Language: Marriage of Actions and Objects. In *Proceedings 11th International Conference on Distributed Computing Systems*, 1991.

[Joh86]   Stephen C. Johnson. Yacc: Yet Another Compiler-Compiler. In *Unix Programmer's Supplementary Documents, Volume 1 (PS1)*, chapter PS1:15. USENIX Association, 1986.

[Lea91]   Doug Lea. Steps Toward an Internal Representation System for the Semantic Analysis of C++ Programs. In *Proc. of OOPSLA*, October 1991.

[Lea94]   D. Lea. ODL: Preliminary Language Report. Technical report, CASE Center, Syracuse University, 1994.

[LFJ+86]   Samuel J. Leffler, Robert S. Fabry, William N. Joy, Hil Plapsly, Steve Miller, and Chris Torek. An Advanced 4.3BSD Interprocess Communication Tutorial. In *Unix Programmer's Supplementary Documents, Volume 1 (PS1)*, chapter PS1:8. USENIX Association, 1986.

[LS86]   M. E. Lesk and E. Schmidt. Lex - A Lexical Analyzer Generator. In *Unix Programmer's Supplementary Documents, Volume 1 (PS1)*, chapter PS1:16. USENIX Association, 1986.

[NC96]   N. Nagaratnam and G. Craig. *Fuzzy-based Dynamic Program Reconfiguration in Distributed Systems*. Florida AI Research Symposium, May 1996.

[NSL96]   N. Nagaratnam, A. Srinivasan, and D. Lea. *Remote Objects in Java(tm)*. IASTED Int'l Conf. on Networks, Jan 1996.

[PCD91]   D. Powell, M. Chereque, and D. Drackley. Fault-Tolerance in Delta-4. *ACM Operating Systems Review*, 25(2):122–125, April 1991.

[Rea91]   J. Rumbaugh and et. al. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[Ros91]   Jim Roskind. `c++grammar2.0`. The source for this grammar is available at USENET archive sites for `comp.lang.c++`, 1991.

[SDP91]   S. Shrivastava, G. Dixon, and G. Parrington. An Overview of the Arjuna Distributed Programming System. *IEEE Software*, pages 66–73, January 1991.

[Sha94]   C. Kevin Shank. *A Computing Framework for Open Distributed Systems*. PhD thesis, Syracuse University, 1994.

[Sta84]   James W. Stamos. Static Grouping of Small Objects to Enhance Performance of a Paged Virtual Memory. *ACM Transactions on Computer Systems*, 2(2), May 1984.

[JBrzezinski 95] J. Brzezinski, et al., "Deadlock Models and a General Algorithm for Distributed Deadlock Detection" *J. of Parallel and Distributed Computing*, Vol. 31, 1995, P.112 - 125

[PBuhr 95] P. Buhr, *µC++ Reference Manual*, U. of Waterloo, Canada, May 1995

[ABurns 87] A. Burns, A. Lister and A. Wellings, *A Review of Ada Tasking*, Springer Verlag, 1987

[KChandy 88] K. Chandy and J. Misra, *Parallel Program Design: a Foundation*, 1988, P.290 - 300

[KChandy 83] K. Chandy, J. Misra and L. Haas, "Distributed Deadlock Detection" *ACM Trans. Comput. Systems*, Vol. 1, No.2, 1983, P.144 - 156

[DChampeaux 93] D. Champeaux, D. Lea and P. Faure, *Object-Oriented System Development*, 1993

[JHogg 92] J. Hogg, D. Lea, R. Holt, A. Wills and D. de Champeaux, "The Geneva Convention on the Treatment of Object Aliasing" *OOPS Messenger,* April 1992

[DLea 94] D. Lea and K. Shank, *ODL: language report,* CASE center at Syracuse Univ., Dec. 1994

[SFerenczi 95] S. Ferenczi, "Guarded Methods vs. Inheritance Anomaly (Inheritance Anomaly Solved by Nested Guarded Method Calls" *ACM SIGPLAN Notice,* Vol. 30, No.2, Feb. 1995, P.49 - 58

[SFrolund 92] Svent Frolund, "Inheritance of synchronization constraints in concurrent Object-Oriented Programming Languages" *Proc. ECOOP 92*

[SFrolund 94] SFrolund and G. Agha, "A Language Framework for Multi-Object Coordination", U. of Illinois at Urbana-Champaign, 1994

[AKshemkalyani 94] A. Kshemkalyani and M. Singhal, "On Characterization and Correctness of Distributed Deadlock Detection" *J. of Parallel and Distributed Computing,* Vol. 22, 1994, P.44 - 59

[SMatsuoka 93] S. Matsuoka and A. Yonezawa, "Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Laguage" *Research Directions in Concurrent Object-Oriented Programming,* MIT Press, 1993, P.107 - 150

[MRosler 89] M. Rosler and W. Burkhard, "Resolution of Deadlocks in Object-Oriented Distributed Systems" *IEEE Trans. Comput.,* Vol. 38, No. 8, 1989, P.1212 - 1224

[BSanders 89] B. Sanders and P. Heuberger, "Distributed Deadlock Detection and Resolution with Probes" *Lecture Notes in Computer Science,* Spring-Verlag, New York/Berlin, 1989, P.207 - 218

[KShank 94] K. Shank, "Diamonds: Architure and Design", Ph.D dissertation, Syracuse Univ. 1994

[MSinghal 89] M. Singhal, "Deadlock Detection in Distributed Systems" *Computer,* Nov. 1989, P.37 - 48

[BStroutStrup 93] B. StroutStrup, *The C++ Programming Language,* 2nd Edition, 1993

[SunMicro 95] SunMicro Lab, "JAVA Language Specification", Oct. 1995

[Tampere 90] "The DisCo Language," Tampere Univ. of Technology, Nov. 1990

MISSION
OF
ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Material Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.